Using Interactive Sketch Interpretation to Design Solid Objects

David Pugh

4 April 1993
CMU-CS-93-147

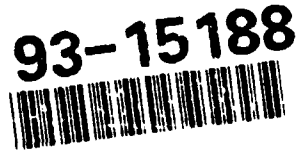
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

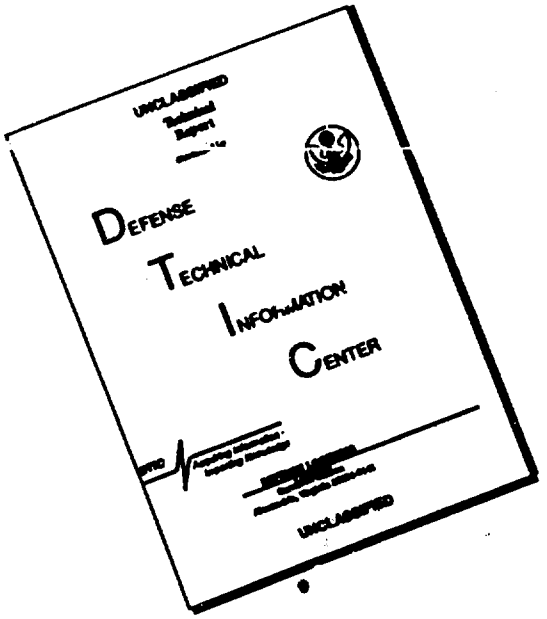
Thesis Committee:
Roger Dannenberg, Chair
Takeo Kanade
Doug Tygar
Rob Woodbury, Department of Architecture

This research was partially supported by a National Science Foundation Graduate Fellowship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. Government.



DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Keywords: Computer aided design, solid modeling, constraint satisfaction, sketch interpretation, line-labeling (for non-trihedral and non-manifold objects)

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Using Interactive Sketch Interpretation to Design Solid Objects

DAVID PUGH

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Roger B. Dannenberg — 4/16/93
THESIS COMMITTEE CHAIR — DATE

[signature] — 4/20/93
DEPARTMENT HEAD — DATE

APPROVED:

R. Rutenbar — 4/28/93
DEAN — DATE

Abstract

Before the introduction of Computer Aided Design and solid modeling systems, designers had developed a set of methods for designing solid objects by sketching their ideas using pencil and paper, and refining these ideas into workable designs. These methods are different from those used for designing objects with a conventional solid modeler. Not only does this dichotomy waste a vast reserve of talent and experience (people typically start sketching from the moment they first pick up a crayon), but it also has a more fundamental problem: designers can use their intuition more effectively when sketching than they can when using a solid modeler.

This dissertation introduces interactive sketch interpretation as a new user interface paradigm for solid modeling systems. Interactive sketch interpretation makes it possible to use the computer as a sketchpad for designing three-dimension objects. The premise behind interactive sketch interpretation is to let the designer change an object's design by modifying a computer generated line drawing of the object. Sketch interpretation maps the designer's changes onto a boundary representation model of the object. The designer can continue the design process by then changing the line drawing of the modified object. This design cycle is highly interactive and, as a result, incorrect interpretations can be easily corrected by the designer.

Viking is a solid modeling system whose user interface is based on interactive sketch interpretation. With *Viking*, the designer can modify his or her design by either changing the line drawing or placing geometric constraints on the object. Sketch interpretation changes *Viking's* model of the object so that it is consistent with the modified line drawing and the geometric constraints placed by the user. The resulting user interface combines the "friendliness" of paper and pencil sketches with the power of traditional solid modeling systems.

Contents

Chapter 1

Introduction

Sketching has long been an important element of the design process. For hundreds of years, people have designed by making quick, abstract drawings or "sketches." Sketching both gave form to embryonic concepts and helped refine these concepts into workable designs. Thirty or so years ago, the arrival of Computer Aided Design (CAD) and solid modeling systems began to revolutionize some aspects of the design process. These programs let designers create a model of a three-dimensional object on the computer. This model can be analyzed in ways that are difficult or impossible without the computer. For example, CAD systems and their associated programs can display realistic images, perform stress analyses, and generate milling machine programs from the computer's model of the object.

Unfortunately, the CAD revolution did not extend to two critical aspects of the design process: exploring new ideas and refining these ideas into workable designs. With current CAD systems, the model typically changes in large, discontinuous steps. The designer is often forced to specify a change completely before he or she has a chance to see how it interacts with the rest of the model. This makes it difficult to do "feedback driven" design, in which the designer uses feedback from one change to guide the next change, on a solid modeler: the magnitude of each change is too large to let the designer use his or her intuition effectively. As a result, designers will often "work out" changes using pencil and paper before making the change on the computer.

The techniques used to design objects on pencil and paper are different from those used to design objects on a solid modeler [30]. Sketching, in this context, is a visual and intuitive process in which a drawing is refined by making small, incremental changes. At each point in the process, the designer uses feedback from one change – the appearance of the modified sketch – to guide the next change. This continual feedback lets the designer use his or her intuition effectively.

This thesis presents a solid modeling system, *Viking*, that uses interactive sketch interpretation to let the user design three-dimensional objects using techniques normally used to create and refine two-dimensional sketches. Interactive sketch interpretation provides a "what you draw is what you get" user interface in which users modify an object by changing its line drawing or geometric constraints. Sketch interpretation, using topology generation and geometric constraint satisfaction, finds a new object that is consistent with changes made by the user. The user can then continue the process of modification and interpretation, gradually transforming a rough sketch into a description of a precisely dimensioned solid object. The resulting user interface combines the power of traditional solid modeling systems with the continuous feedback of sketching.

1.1 Designing with solid modelers vs. designing with sketches

The design "process" can be thought of as a cycle in which the designer interprets the design and then modifies it based on his or her interpretation. This cycle is repeated until the designer is satisfied with the design. This description of the design process applies equally well to designers using a solid modeler and designers making pencil and paper sketches. The difference between these two cases lies in the ways each represents the design internally, how information is displayed to the designer and the ways in which the designer manipulates the design.

Using a solid modeler for design is similar, in some respects, to the dialogue between a master and apprentice: the master tells the apprentice what to do, the apprentice does the work and presents the results to the master. As in the classic tradition, the master must carefully phrase his or her requests since apprentices are famous for doing what they are told in unexpected ways. Unlike the classic apprentice, however, a computerized apprentice cannot, currently, learn from his or her mistakes and it is the master who must learn to accommodate the foibles of the apprentice.

If using a solid modeler is like working with an apprentice, then paper and pencil sketching is like working alone. Working alone can have some advantages: some tasks are easier to do than they are to explain. Unfortunately, working alone also means that the designer has to do all the work, including the boring or tedious parts.

1.1.1 Solid modeling systems

The ways in which a designer can modify a design on a solid modeler fall into four broad categories: direct generation, constructive solid geometry (CSG), profile manipulation and deformable surfaces. Direct generation lets the designer create simple geometric primitives (cubes, spheres, cones, etc.) by specifying their type and dimensions. CSG lets the designer use Boolean operations to create complex objects by combining simpler ones (for example, create a "lens" by finding the intersection of two offset spheres). Profile manipulation lets the designer create solids by sweeping two-dimensional profiles through space (for example, create a torus by sweeping a circular profile along a circular track). Deformable surfaces let the designer change an object's shape by stretching, twisting and bending it [9].

Strengths

One of the greatest strengths of solid modeling systems is the help they give the designer in visualizing the object. Because they have a three-dimensional model of the object, solid modeling systems have enormous flexibility in how they display the object. For example, designers can "see" their object from any viewpoint or watch an animated sequence of a mechanism in action. The wide variety of ways to view the object lets the designer pick the best display for understanding the design and how it might work in the "real world."

Another advantage of using solid modelers is that their object description can be analyzed numerically. This, for example, lets designers calculate the stresses within the object and use this information when modifying the object. As a result, designers have more information on which to base design decisions and, therefore, can work faster and with greater confidence.

It is easy to use a solid modeling system to create complex regular objects. These objects, such as a ventilation grill or threaded bolt, have shapes that can be described algorithmically. Creating

these objects by hand is difficult and time consuming.

Weaknesses

One problem with solid modeling systems is that they can be difficult to use. In part, this is because many of the more powerful operations do not have "real world" analogs and, therefore, are hard to use intuitively. For example, one of the easiest ways to create a hexagonal slab in the "real world" is to cut the four corners off a rectangular slab. However, one of the easiest ways to create a hexagonal slab in a solid modeler is to use Constructive Solid Geometry to find the intersection of three correctly oriented rectangular slabs [3] (it is also possible to use four CSG "cuts" to remove the four corners off a rectangular slab).

Another problem with solid modelers is that they often present the designer with a "catch-22" situation. The designer must precisely describe the geometry of a change before the solid modeler can make it; but the designer often cannot describe the change precisely until he or she has had a chance to see how it interacts with the rest of the design. In many cases, the designer will not know how to describe a change precisely until long after the change has been made.

1.1.2 Paper and pencil sketches

In comparison, paper and pencil sketches are the essence of simplicity: the design is represented by and displayed as a two-dimensional drawing. The designer manipulates the design by modifying the drawing. Despite this simplicity, sketches perform two valuable functions in the design process. Sketches help the designer visualize the three-dimensional object represented by the drawing and they provide a framework in which the designer can split the problem into manageable chunks.

The drawings used in the design process range from abstract sketches, to accurate mechanical drawings, to realistic illustrations. Different types of drawings are used for different design tasks. This thesis will concentrate on how sketches are used in design. While accurate or realistic drawings have their uses, they are not relevant to integrating sketching and solid modeling since an integrated system can easily produce accurate and realistic images from the three-dimensional object description

Strengths

Designers who create a sketch have complete control of the "information flow." They are never forced to work in a particular order, or provide information until they are ready to give it. Sketches let designers pick how much detail and precision to use. And, as designers learn more about the design, they can modify or refine sketches to make them more detailed or accurate.

Sketching also provides an intuitive environment in which to work. When sketching, the designer is modifying the sketch directly and there is no uncertainty about how any change will modify the sketch's appearance. In contrast, designers using a solid modeling system modify the underlying three-dimensional object description, which might change the display in unexpected ways.

Sketching is also a good way to explore new ideas because sketches are refined incrementally. If a designer wants to try a new idea, he or she can first sketch the broad outlines of the idea and

then refine the sketch as the idea develops. If the idea turns out to be a bad one, it can be abandoned before the designer has invested significant effort.

Sketches can be used to represent objects with a wide range of complexities. Simple sketches are easy to create. Complex sketches typically the result of progressively refining simpler sketches. And, while it may take a long time to sketch a complex object, it can easily take as long to create a model of the object using a solid modeler. This is especially true when the object's geometry is irregular.

Weaknesses

For all their usefulness, sketches are limited. Although they can help the designer visualize the design, solid modeling systems do a much better job. Also, there is no way to do numeric analysis on a sketch besides transferring it, by hand, to a solid modeler. For simple designs, these are not significant problems: designers may not need much help to visualize simple objects and experience can help a designer estimates the properties, such as stresses, that could be calculated using numeric analysis.

In addition, the process of creating a sketch can be exceptionally tedious. Every line must be drawn by hand and may, as the design changes, have to be erased and redrawn many times. Many of the operations that are trivial on a solid modeler are difficult to do on a sketch. For example, changing the viewpoint means redrawing the entire sketch.

1.2 *Viking*: combining solid modeling and sketching

The strengths of sketching seem to complement the weaknesses of solid modeling and vice versa. Therefore, combining elements from both sketching and solid modeling defines a new design methodology that is potentially better than either sketching or solid modeling alone. This design methodology is implemented in *Viking* by combining direct manipulation, geometric constraint satisfaction, topology generation, and three-dimensional feedback into a single, integrated design system.

Direct manipulation is fast but imprecise. Geometric constraint satisfaction lets the designer precisely specify a desired geometry in a convenient fashion but typically requires initial values that are close to a solution. The strengths of each technique complements the weaknesses of the other: direct manipulation lets the designer put vertices in "about" the right position and constraint satisfaction lets him or her put vertices in exactly the right position. Topology generation lets *Viking* generate a fully specified solid or thin-shell object description from the line drawing without having the designer explicitly specify the object's faces. This, in turn, lets *Viking* provide all the visualization aids (i.e. three-dimensional feedback) that a conventional solid modeler provides while retaining the "feel" of creating a pencil and paper sketch.

1.2.1 Establishing a design dialog

Viking uses sketches to establish a "dialog" between the designer and the computer. Sketching is an interactive process in which sketches serve as additional "working memory" for the designer. Sketches give designers a forum in which they can easily make a change and "see" the results

immediately. This lets the designer evaluate each change and protects it from the vagaries of human memory.

These sketches, however, are more than just line drawings. In *Viking*, "sketches" are three-dimensional entities that describe both the appearance of the object and the geometric constraints specified by the designer. Despite this complexity, however, the designer must be able to easily modify *Viking's* sketches. Otherwise, he or she will resort to easily modified pencil and paper sketches, defeating the purpose of combining sketching and solid modeling.

1.2.2 Capabilities

Viking lets designers change a sketch by:

- drawing new edges or vertices,
- deleting existing edges or vertices,
- hiding or exposing individual line-segments,
- defining relationships between vertices and edges, and
- dragging edges or vertices while maintaining previously defined geometric relationships.

Sketch interpretation is used to map these changes onto *Viking's* internal model of the object being designed. This interpretation process takes a variety of forms. It can be an integral part of sketch manipulation, as when using preferred directions (see Section 2.1.1) to position a vertex in three space. It can also be a postscript to a change, as in solving for a new vertex geometry after placing a new constraint. In either case, the essence of sketch interpretation is to let the designer express any desired change in a natural and intuitive fashion and automatically apply the change to the internal model.

Designers can also modify his or her "environment" by:

- defining preferred directions (see Section 2.1.1),
- defining a cutting plane (see Section 2.1.1),
- moving the cutting plane through the object,
- setting a preferred object type (see Section 3.3.1),
- grouping vertices into components (see Section 2.1.3),
- changing his or her viewpoint, and
- modifying the display parameters.

These changes do not modify the object directly. Instead, they modify either the way that the designer's actions are interpreted or the way that the object is presented to the designer.

1.2.3 An example of using *Viking*

Suppose you wish to create a model of a cube with a notch cut through it. One way to do this task, using paper and pencil, is to:

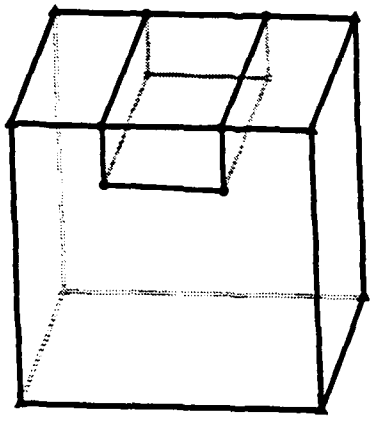


Figure 1-1a: "Draw in" a notch.

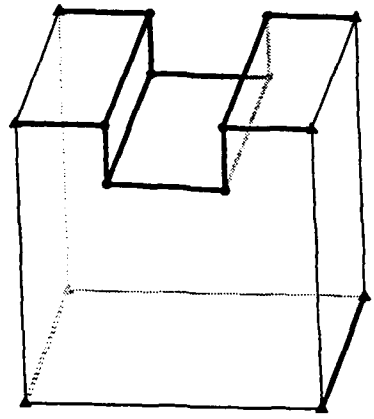


Figure 1-1b: Make it look right.

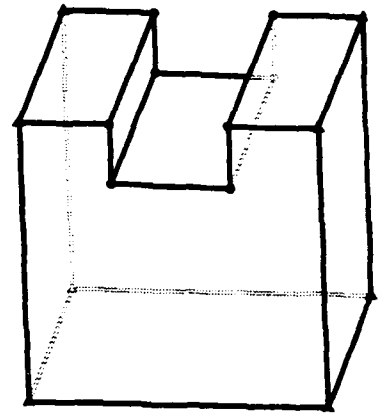


Figure 1-1c: Find an interpretation.

- sketch a cube,
- add the lines forming a notch across the top of the cube,
- erase unwanted lines, and
- redraw the notch's lines so that they are solid (e.g. visible) or dashed (e.g. hidden) as appropriate.

One way to do this task, using a solid modeler, is to:

- create a cube,
- create a rectangular block that is longer and narrower than the cube,
- position the rectangular block along the top of the cube so that the two objects intersect, and
- do a CSG "cut" operation: find a new object that corresponds to the volume enclosed by the cube and not enclosed by the block.

Performing this task on *Viking* is similar to performing it on pencil and paper, although the results are the same as those for doing it with a solid modeler. Figures 1-1a, 1-1b and 1-1c show a sequence of "snap shots" taken while the user is creating a notched block. Starting with a unit cube, the user has, in Figure 1-1a, drawn in the lines that form the notch. The user has, in Figure 1-1b, cleaned up the image. Line-segments that would be visible if the notch existed have been made visible and the unwanted edges have been deleted. Figure 1-1c shows the first interpretation found for the image in Figure 1-1b. In this interpretation, every edge is adjacent to two faces and the vertices that form the notch have also shifted slightly to satisfy the implicit constraint that every face is a planar polygon. This object can be moved and rotated and the drawing will update in the expected manner: line-segments that should be hidden in a different display will be hidden. I needed less than 30 seconds to create the object shown in Figure 1-1c using the procedure described above.

Viking gives designers a new way of using the computer to design solid objects. With *Viking*, a designer can sketch the object he or she wishes to create and modify the object by changing the sketch. This design methodology is most useful when an object's appearance is important or when

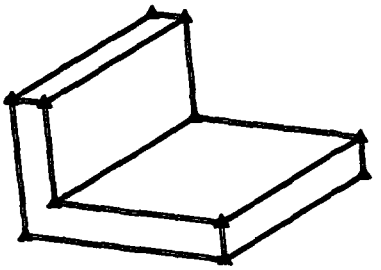


Figure 1-2a: Bracket.

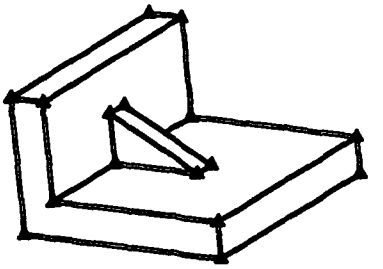


Figure 1-2b: Reinforced bracket.

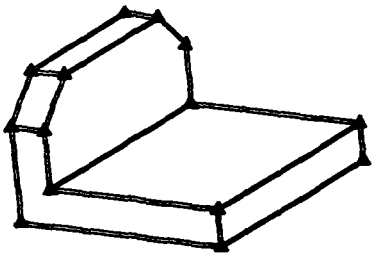


Figure 1-2c: Trimmed bracket.

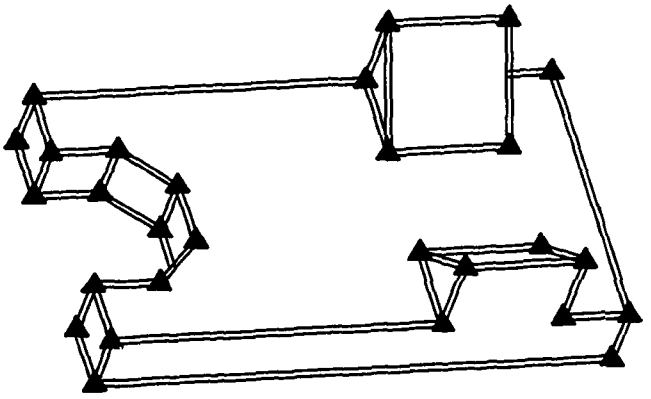


Figure 1-3a: Widget (28 vertices).

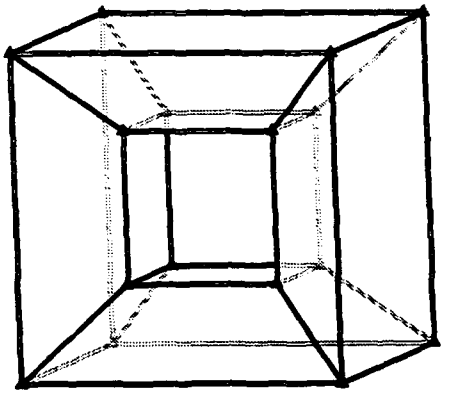


Figure 1-3b: "Tesseract" (16 vertices).

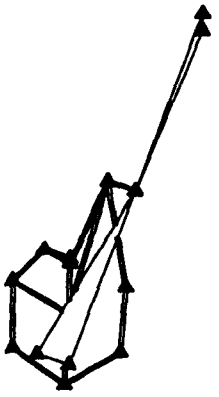


Figure 1-3c: Windmill (16 vertices).

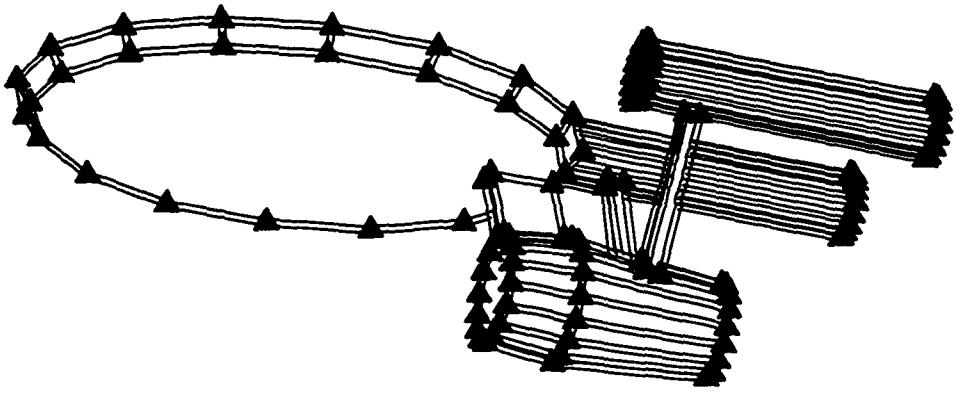


Figure 1-3d: Starship (175 vertices).

the designer can predict an object's behavior from its appearance. In either case, the design process is largely a matter of the designer using his or her intuition to correct things that look "wrong."

The design methodology supported by *Viking*, however, is useful even when the designer is not working intuitively. In particular, *Viking* lets designers use a single technique to modify the object being designed in a wide variety of ways. For example, the simple bracket in Figure 1-2a has been modified in two different ways to create Figures 1-2b and 1-2c. In Figure 1-2b, the designer has added a reinforcing webbing. In Figure 1-2c, the designer has trimmed the upper corners. In both cases however, the same basic technique was used: draw the changes in, find an interpretation and use geometric constraints to do the final positioning.

Although *Viking* was never intended to be a "stand-alone" system, it is powerful and flexible enough to create a wide variety of objects, as shown in Figures 1-3a through 1-3d. I needed about 6 minutes to design the object in Figure 1-3a. Figures 1-3b and 1-3c both took about 4 minutes to complete. Figure 1-3b is interesting because, by hiding some line-segments and exposing others, it is possible to define three distinct solid objects that have the same vertex geometry. Figure 1-3c demonstrates using *Viking* to design non-manifold objects. The object in Figure 1-3d is the most complex object that I have created using *Viking*.

1.2.4 Limitations

Viking has several significant limitations. Many are simply the result of not having enough time to implement all of the desired functionality (for example, *Viking* cannot calculate the volume of an object under construction). Others are the results of the algorithms used for various tasks. The effect of these limitations is to restrict *Viking* to designing objects that meet the following criteria:

- no vertex has more than four adjacent edges,
- no edge has more than two adjacent faces,
- every edge is a straight line, and
- every face is a planar polygon.

It should be possible to overcome most of these limitations in future versions of the program (see Section 7.2).

Topology generator

Viking's topology generator uses a form of line-labeling [7, 17] to generate a topology from a line drawing. This algorithm depends on the line drawing being a "general view:" one in which a small change in the observer's viewpoint will produce a correspondingly small change in the line drawing [24]. In *Viking*, general views have the following properties:

- every face is drawn as a closed polygon with a non-zero area,
- every edge is drawn as a line with a non-zero length, and
- if two adjacent lines are parallel, then the corresponding edges are parallel.

If a line drawing does not have these properties, then *Viking* cannot use it to generate a topology for the object it represents.

Another limitation in *Viking's* topology generator is in the type of faces it can generate. In particular, *Viking* can only generate faces that have the following properties:

- no vertex or edge is repeated, and
- there are no "bridging" edges.

The first property makes it impossible for *Viking* to generate faces with internal holes. The second property is a side-effect of the constraints used to test the validity of an interpretation. A "bridging" edge is one that extends across a face and between two of the face's vertices. Faces containing bridging edges are rejected because the edge must lie in the plane defined by the face and, therefore, cannot satisfy the *in front of* or *behind* constraints used to verify the interpretation's validity.

Constraint solver

Viking's constraint solver, like most others, uses the current position of the vertices as a starting point when searching for a geometry that satisfies the constraints. This technique is most effective when there is a "nearby" geometry that satisfies the constraints. It is slow and unreliable when there is no "nearby" geometry that satisfies the constraints. This means that *Viking's* constraint solver cannot be used to "unfold" a flat two-dimensional sketch into a three-dimensional object description. Instead, *Viking's* users must create a three-dimensional sketch in which the vertices are drawn so that they are close to their intended x, y and z coordinates. Fortunately, *Viking's* user interface provides the tools to make it relatively easy to position vertices in space (see Section 2.1.1).

1.3 Success criteria

This thesis describes a new user interface paradigm – interactive sketch interpretation – for computer aided design and its implementation in *Viking*. Interactive sketch interpretation makes it possible to design solid objects using techniques normally associated with making pencil and paper sketches. It is not the intent of this thesis to make any of the existing solid modeling techniques obsolete. It is, instead, intended to start exploring the potential of integrating sketching into computer aided design. Therefore, this thesis will be considered a success if it:

- combines direct manipulation, geometric constraint satisfaction, topology generation and three-dimensional feedback into a solid-modeling user interface,
- describes the underlying algorithms so that other researchers can implement and experiment with interactive sketch interpretation, and
- shows that the combined user interface can be used to create precisely dimensioned three-dimensional models.

1.4 Related work

Viking combines several different ideas to create a user interface based on direct manipulation, sketch interpretation and constraint satisfaction. Although other systems have used one or more of these ideas, *Viking* is the first system to combine all of them into a single system for doing solid modeling. Direct manipulation and constraint satisfaction were both part of Sketchpad [28]. Sketch interpretation systems using line-labeling have been around for over 20 years [7, 17]. Three-dimensional feedback has been an integral part of solid modeling since its inception.

1.4.1 Direct manipulation

Almost every design system uses some form of direct manipulation. These systems let the user position vertices and objects by pointing and dragging with the mouse (or other pointing device). Direct manipulation has also been used to change the view transform [5], or arbitrary parameters [2].

1.4.2 Sketch interpretation

Traditional image interpretation (generating useful information from a camera image) is different and more complex than the type of sketch interpretation done by *Viking*. In particular, image interpretation must deal with noise, shadows, different surface colors and textures, etc. In contrast, *Viking's* sketch interpretation algorithm only has to process a "noise-free" line drawing in which hidden line-segments and all vertices are explicitly represented.

This problem domain is similar to that used by Guzman [14], Huffman [17] and Clowes [7] for their work on line-labeling. Huffman-Clowes line-labeling reduces the problem of determining whether a line drawing represents a trihedral object[1] to one of finding mutually consistent labelings for each intersection in the line drawing. This basic algorithm has been extended since then to a wide variety of objects and line drawings: scenes containing shadows [29], line drawings in which hidden lines are visible [24], objects with curved surfaces [6], and thin-shell objects [18]. Also, Waltz [29] introduced an efficient algorithm for filtering out obviously inconsistent labelings, and Sugihara [27] developed a way to test the feasibility of an interpretation by solving a linear programming problem. *Viking* extends line-labeling to non-trihedral objects and potentially to objects in which edges can be adjacent to three or more faces.

Viking's algorithm for topology generation is view-dependent in that it finds a topology that is consistent with a line drawing, provided the line drawing corresponds to a general view. There is another family of view-independent algorithms developed by Hanrahan [16] and Courter [8], among others, that can be used to generate surface topologies. These algorithms generate a surface topology for an object by finding the planar embedding of the object's vertex-edge graph.

Leclerc and Fischler [19], expanding on work done by Marrill [20], have developed an algorithm for generating depth information for a line drawing. This algorithm generates a surface topology from a line drawing and then finds z-coordinates for the points in the line drawing that minimize the:

[1] One in which each vertex is adjacent to exactly three faces.

- standard deviation between the angles of all lines radiating from each point, and
- non-planarity of all of the faces.

This work is still in the early stages of development, so it is not clear whether this algorithm will work for a wide variety of objects.

1.4.3 Geometric constraint satisfaction

Viking lets the user define geometric relationships within their models. The problem of finding a geometry that satisfies these relationships is, however, not easy. Non-linear equations are needed to model all but the simplest constraints and solving systems of non-linear equations is known to be a hard problem [13].

Viking reduces the problem of finding a geometry that satisfies the constraints to the following non-linear satisfaction problem:

Solve for $\vec{x}$ such that:

$$\begin{aligned} h_i(\vec{x}) &= 0 \quad (0 \leq i \leq |\vec{h}|) \\ g_j(\vec{x}) &\geq 0 \quad (0 \leq j \leq |\vec{g}|) \end{aligned}$$

Other algorithms for constraint satisfaction use a similar reduction. DeltaBlue [10] adds a constraint hierarchy in which constraints are ranked according to their importance and the solver can violate a constraint in order to satisfy a more important constraint. Witkin's differential constraint [12] adds an explicit objective function that is minimized.

Simultaneous constraint solvers

One way to solve a system of equations is to simultaneously manipulate all of the variables. All of the techniques for doing this can be summarized as follows:

Given $\vec{x_0}, \vec{h}$, and $\vec{g}$

$\vec{x} \Leftarrow \vec{x_0}$
Loop:
$\quad \vec{x} \Leftarrow \vec{x} + \vec{\delta}(\vec{x}, \vec{h}, \vec{g})$
Until $\vec{x}$ satisfies the constraints.

In other words: starting from an initial point $\vec{x_0}$, use the function $\vec{\delta}$ to modify the current position, $\vec{x}$, until a solution is found. The differences between these techniques lie in how they use to modify the current position and the types of constraints that they allow.

One of the problems with all of these techniques is that they depend on having $\vec{x_0}$ be "close" to a solution. If the constraint solver does not find a solution, then either $\vec{x_0}$ was not close enough to a solution or the constraints were mutually inconsistent. Unfortunately, there is no reliable way to determine which problem caused the constraint solver to fail.

All of the algorithms for simultaneously solving systems of non-linear equations use some form of gradient descent to modify the current position. This is most clearly seen in the Newton-Raphson algorithm [22] where, at each iteration, the current position is modified by $-\vec{h}(\vec{x})(\nabla\vec{h}(\vec{x}))^{-1}$. Unfortunately, this has problems of its own:

- The constraint equations might contain singularities.
- The derivatives at $\vec{x}$ might be 0.

The first problem can be avoided by picking an initial point that is closer to a solution than it is to a singularity. The second problem can be managed by using another technique, such as simulated annealing [15], when needed.

Other constraint solvers

Another way to solve a system of equations is to manipulate only a few of the variables at a time. These solvers, called incremental solvers, generates a dependency graph for the variables (e.g. "t depends on z, which depends on y and x"), and then solves for each variable after solving for the variables on which it depends. This type of constraint solver works well when that dependency graph is acyclic. They can, however, iterate indefinitely if the constraint's dependency graph contains cycles. DeltaBlue [10] is an example of an incremental solver.

1.4.4 Other drawing or computer aided design systems

There are several drawing or computer aided design systems that, like *Viking*, combine direct manipulation with geometric constraint satisfaction. The most obvious difference between these systems is that some can be used to design three-dimensional objects and others are limited to creating two-dimensional drawings. Less obvious differences are their algorithms for solving systems of constraints, and their user interface for manipulating the object.

Sketchpad

Ivan Sutherland's Sketchpad [28] was a two-dimensional design system in which users could "draw" on the screen using a lightpen. Buttons on a separate control panel let the user switch between drawing lines, moving points and placing various types of constraints. For example, the user could constrain a point to lie on a circle by picking it up with the lightpen while holding the "move" button, dragging it to a position on the circle's circumference and giving a termination flick. Sketchpad could also be used for analysis. In one example, Ivan Sutherland modeled a truss and used Sketchpad's constraint solver to calculate the reaction forces in response to loads "placed" with the lightpen.

Sketchpad was a product of a radically different computing environment. Back in 1963, computer graphics were done on oscilloscopes, menus were what you found in a restaurant and mice were furry. As a result, Sketchpad may seem slow and clumsy by today's standards. Do not, however, let this obscure the fact that Sketchpad introduced many key ideas – direct manipulation, pointing devices, constraint satisfaction, etc. – that were used by later systems.

Juno

Brian Nelson's Juno [21] is a constraint-based system for two-dimensional graphic design. Juno's user interface gives the user two "views" of the drawing: the drawing itself and a textual description of a procedure that generates the drawing. The user can either modify the procedure or

change the drawing, in which case Juno automatically updates the procedure so that is consistent with the modified drawing.

Juno uses the Newton-Raphson algorithm [22] to find a solution to the drawing's constraints. Juno uses the current geometry as a starting point for the Newton-Raphson algorithm. So, if no solution is found (or the wrong solution is found), the user can modify the drawing so that it is "closer" to the desired solution. The constraints themselves can take a wide variety of forms including parallel, equal length, and orthogonal. These constraints are represented only in the procedure that describes the drawing. They are not "visible" in the drawing except through their effect on the drawing's geometry.

Gargoyle

Eric Bier's Gargoyle [1] introduced a new design methodology: snap-dragging. Gargoyle lets the user activate various constraints (for example, activate a "distance = 1" constraint). Gargoyle then draws construction objects on the screen corresponding to the active constraints (for example, if the "distance = 1" constraint was activated, Gargoyle would draw unit circles around all of the points in the drawing). The cursor then snaps to these construction objects, establishing a precise geometric relationship. In practice, using Gargoyle is similar to, but considerably easier than, traditional mechanical drawing. Gargoyle could be used for both two-dimensional and three-dimensional design.

Gargoyle is different from the other systems described in this section in that it "forgets" geometric relationships once they have been established. If the user wants to modify a design by moving a point, he or she must manually re-establish all of the affected relationships. As a result, Gargoyle's constraint solver only needs to solve for the position of one point at a time. It needs, however, to do this very quickly: Gargoyle provides continuous feedback on the location that the cursor would snap to.

Briar

Michael Gleicher's Briar [11] is a two-dimensional design system that extends the snap-dragging methodology by adding persistent constraints. Just as with Gargoyle, the user can activate the desired constraints and the cursor will snap to positions that satisfy them. The user may also use the cursor to drag points from one position to another and Briar will automatically adjust the drawing's geometry to satisfy the previously established constraints.

Briar uses a differential constraint solver [12] to maintain the constraints. Differential constraint solvers rely on having all of the constraints satisfied initially and trying to find a different geometry in which an objective function is minimized and constraints are still satisfied. For example, if the user drags a point in Briar with the mouse, Briar will move the point so that the constraints remain satisfied while minimizing the distance between the point and the mouse's current location. The difficulty behind this approach to solving systems of constraints is that it is most effective when all of the constraints are initially satisfied. Fortunately, with the snap-dragging methodology, the active constraints are satisfied whenever a new point or piece of geometry is added to the drawing.

Converge

Steven Sistare's Converge [26] is a three-dimensional constraint based design system. In many respects, Converge and *Viking* have similar objectives: three-dimensional design. As a result, these two systems have a great deal in common. Both systems, for example, let the user dynamically adjust the viewpoint, drag vertices while maintaining the constraints, and group vertices into distinct components.

There are, however, some differences between the two systems. Converge is a notable for having an efficient and robust constraint solver but it can only be used to design an object's geometry. Unlike *Viking*, it does not let the user specify an object's topology. As a result, it cannot generate hidden line displays (which are useful for helping the designer visualize the object's three-dimensional structure).

1.5 Thesis outline

This thesis is divided into four distinct parts. Chapters 1 and 2 describe the interactive sketch interpretation design methodology and its implementation in *Viking*. Chapter 2 describes *Viking's* user interface and provides two additional examples of using *Viking* to design three-dimensional objects.

Chapters 3, 4 and 5 present the algorithms used to perform topology generation and constraint satisfaction. Chapter 3 describes the algorithm used to generate a surface topology from a line drawing. Chapter 4 describes the algorithms used to generate and solve systems of non-linear equations. Together, chapters 3 and 4 describe the algorithms used to generate a three-dimensional object description from a modified line drawing and a system of geometric constraints. Chapter 5 describes the algorithm used to create *Viking's* intersection library (see Chapter 3 and Appendix D).

Chapters 6 and 7 describe *Viking's* performance and draw conclusions about the interactive sketch interpretation design methodology and *Viking's* implementation of it in particular. Chapter 6 plots the time required to find a surface topology or solve for a vertex geometry for objects with 9 to 100 vertices. Chapter 7 contains the conclusions as well as future plans.

The appendices contain technical data. Appendix A is a glossary. Appendix B is a description of all the different actions that the user can perform. Appendix C contains the equations used to represent the different geometric constraints. Appendix D contains the intersection library for all intersections of 2, 3 and 4 lines. Appendix E contains the written feedback from a group of students who used *Viking* to create a cuboctahedron as an assignment for a geometric modeling class.

Chapter 2

The *Viking* solid modeler

Viking is a solid modeling system that combines sketching with sketch interpretation. Designers can "sketch" with *Viking* in much the same way that they sketch with pencil and paper. The critical difference is that, in *Viking*, sketch interpretation maps changes to a sketch onto a description of the three-dimensional object. For most changes, deducing an appropriate change in the object description is trivial. For example: if the user erases a line, delete the corresponding edge. With other changes, such as making a line-segment visible, there is no obvious corresponding change in the object description. In these cases, *Viking* used heuristics to generate a new object description that is both reasonable and consistent with the modified line drawing.

The task of generating an interpretation of a line drawing is split into two parts: finding a surface topology and solving for a vertex geometry. The first part is done by generating surface topologies that are consistent with the line drawing until an acceptable one is found. The second part is done by using a geometric constraint solver to find a vertex geometry that satisfies a set of constraints that are either derived from the line drawing or placed by the user. The surface topology and vertex geometry combine to form a three-dimensional object description that is consistent with both the line drawing and the constraints.

2.1 *Viking's* user interface

Figure 2-1 shows *Viking's* display after creating an equilateral triangle. The left window shows a line drawing of the underlying object description and the upper center window shows the view transform used to generate the line drawing. Both windows let the user directly modify their contents. The user can, for example, move a vertex by dragging it to a new location with the mouse. The user can also dynamically change the view transform by dragging the mouse across the orientation triad, causing the view to rotate about an axis perpendicular to the mouse's motion. This particular method of manipulating a view transform is also called a continuous XY controller [5].

The line drawing displays more than just an object's shape. Thick, thin and double lines respectively correspond to edges adjacent to zero, one and two faces in the object description. Circles correspond to vertices that the constraint solver can move when solving for a vertex geometry. Triangles correspond to vertices whose positions are considered fixed constants by the constraint solver. A variety of symbols is used to indicate constraints. Distance constraints, for

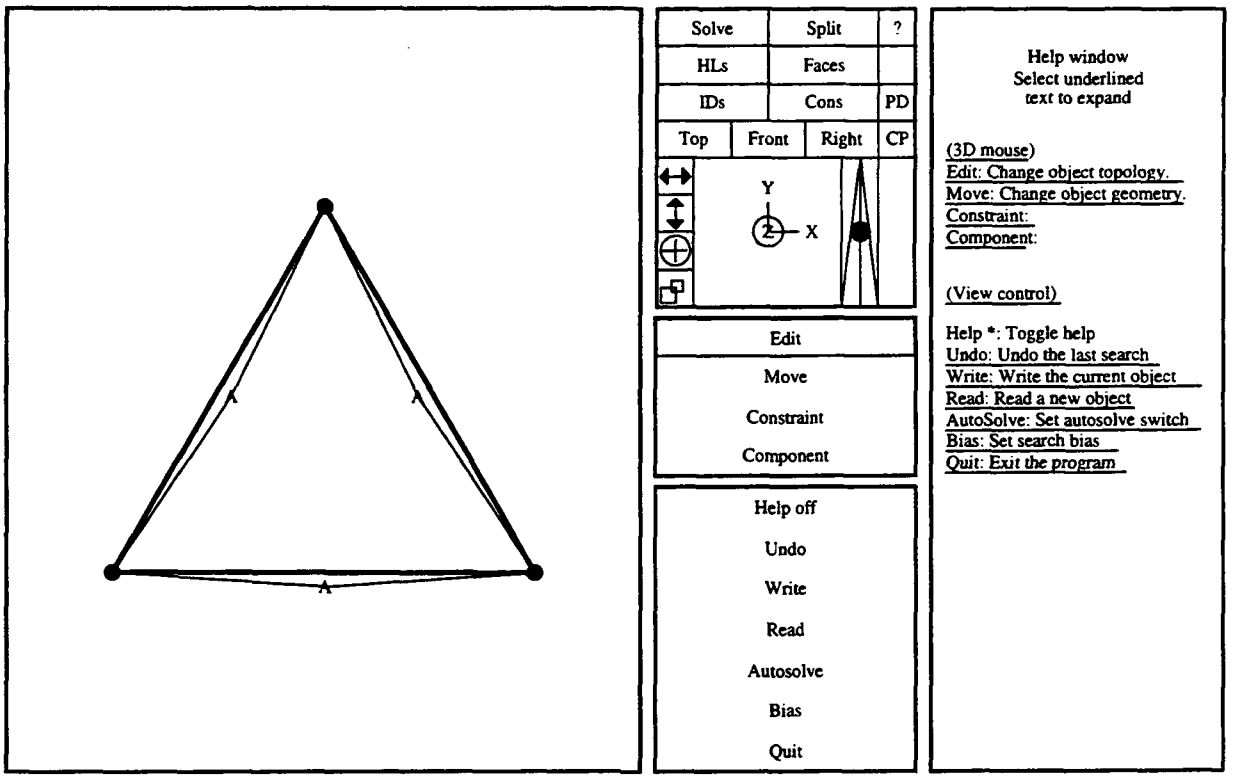


Figure 2-1: *Viking's* display.

example, are shown by thin, bent lines. In Figure 2-1, the "A" symbol at the bend indicates that all three sides of the triangle have the same length.

2.1.1 Sketching in three dimensions

Traditional sketches are two dimensional even though they may represent three dimensional objects. In *Viking*, however, sketches are three-dimensional entities in which every vertex in the sketch has a position in three-space. This helps the user since a three-dimensional "sketch" can be examined from any angle or viewpoint. But it also means that each vertex must be correctly positioned in three space.

It, however, is not necessary to specify a vertex's position precisely. Two-dimensional sketches are often drawn with little attention to precision since a sketch that is approximately "right" is both sufficient for much of the design process and easier to create than one that is precisely drawn. Three-dimensional sketches are similar: they do not have to be accurate to be useful. *Viking* lets the user "sketch" in three dimensions by making it easy for him or her to specify the approximate position of a vertex in three space when it is added to the sketch.

Geometric constraints are not, by themselves, a good mechanism for specifying approximate vertex positions. In part, this is because the constraint solver works best when all vertices are near a solution. Relying on the constraint solver to move a vertex a significant distance is, at best, time consuming and often results in unexpected solutions (assuming a solution is found). A more fundamental problem with using constraints for rough positioning, however, is their precision. Often, users do not know the precise location of a vertex until late in the design process. Using

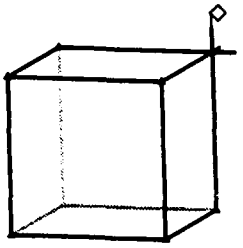


Figure 2-2a: Default preferred directions for a vertex.

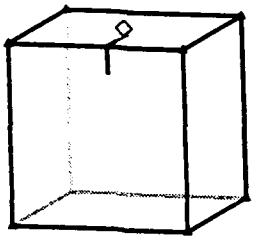


Figure 2-2b: Default preferred directions for an edge.

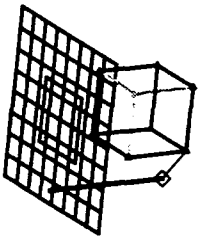


Figure 2-3: Using cutting planes.

constraints to position a vertex before the user knows its precise location is time consuming since the user will have to change the constraints later, when the precise dimensions are known. It can also be intimidating: people do not like answering questions until after they know the answers.

Viking uses three basic techniques to let the user "sketch" in three dimensions. When no other information is available, *Viking* uses a simple rule when drawing edges: both of the edge's vertices have the same z-coordinate in the display's coordinate space. For many cases, such as drawing a short edge from an existing vertex, this is sufficient. And, for situations where this is not sufficient, *Viking* provides two additional mechanisms to let the user specify the location of a vertex in three dimensions: preferred directions and cutting planes.

Preferred directions

Preferred directions are three-dimensional vectors. When the user draws an edge, *Viking* draws short lines parallel to each preferred direction at the new edge's origin. As the user moves the mouse, *Viking* projects the edge's endpoint onto the closest preferred direction. In Figure 2-2a, for example, the edge's endpoint is being projected onto a preferred direction that is parallel to the vertical edge adjacent to the starting vertex. In Figure 2-2b, the edge's endpoint is being projected onto a preferred direction that is perpendicular to the starting edge and lies in the plane of the cube's top face.

Preferred directions can be defined in two ways. First, the user can define vectors in object space, such as the x, y and z axes, for preferred directions. Any new edge, no matter where it is drawn, will be able to use these preferred directions. Second, the user can activate automatic preferred directions. In this case, *Viking* automatically defines preferred directions depending on the context in which the user started to draw the new edge. If the user is drawing an edge from an existing vertex, then the preferred directions are defined to be parallel to each of the edges radiating from the vertex. If the user is drawing an edge from an existing edge, then one preferred direction is defined to be parallel to the edge and, for each adjacent face, a preferred direction is defined to lie in that face's plane and be perpendicular to the edge. If these rules generate only one preferred direction, then two preferred directions are added that are perpendicular to both the original preferred direction and each other. If two preferred directions were generated, then a third preferred direction is added that is perpendicular to the first two preferred direction.

Cutting planes

A cutting plane is a plane defined in object space. Cutting planes are a tool for both positioning a vertex in three dimensions and helping the user visualize the object's three-dimensional structure. The user can position a vertex in three dimensions by moving it parallel to the cutting plane or parallel to the cutting plane's normal. For example, in Figure 2-3, the new vertex is being positioned so that it is the same distance from the cutting plane that the starting vertex is.

The user can manipulate the cutting plane by moving it parallel to its normal, changing the orientation of its normal, and controlling the way in which it is displayed. The user can, among other things, make the cutting plane opaque or translucent, highlight the intersection of the cutting plane with the object, show the orthogonal projection of the object onto the cutting plane, and draw height poles between each vertex and the cutting plane.

2.1.2 Sketch interpretation

Viking's implementation of interactive sketch interpretation uses two distinct data-structures: one holds the current object description and the other holds the line drawing displayed to the user. When the user modifies the line drawing, the changes automatically propagate to the current object description to maintain consistency between the two data-structures. When the user changes the viewpoint, the line drawing is recreated from the new view transform and the current object description.

Viking generates a new object description whenever the user makes a change that cannot be propagated to the object description automatically. *Viking* splits the task of generating a new object description into two parts: finding a surface topology that is consistent with the line drawing and solving for a vertex geometry that satisfies the object's implicit and explicit constraints. Together, the surface topology and the vertex geometry completely describe a three-dimensional object. The new object description is consistent with both the line drawing created by the user and any geometric constraints he or she may have specified.

Viking uses arc-labeling [23], an extension of Huffman-Clowes line-labeling [7, 17] to non-trihedral vertices, to generate a surface topology from a line drawing and an old object description. The surface topology defines a set of faces that are consistent with the line drawing. Since line drawings can have many different interpretations, *Viking* uses a cost heuristic to seek out the more desirable interpretations first. *Viking* generates surface topologies in order of increasing cost, where the cost is based on several factors, including:

- how similar the surface topology is to the current object's surface topology and
- if the user has given a preferred object type, how close the surface topology is to the user's preferred type.

Surface topologies are generated until the user either accepts one or aborts the search. In my experience, the desired surface topology is normally the first surface topology found.

Once an acceptable surface topology has been found, a non-linear constraint solver finds a vertex geometry that satisfies a system of geometric constraints. These constraints fall into three categories:

- World: every face is a planar polygon.

- Image: hidden lines are behind obscuring faces and lines.
- Explicit: constraints explicitly defined by the user.

The first two types of constraints are implicit constraints since they are automatically generated by *Viking*. World and explicit constraints are always part of the system of equations used by the constraint solver. Image constraints are only used when finding a vertex geometry after finding a new surface topology for the object.

The constraint solver uses an algorithm developed by Bullard and Biegler [4]. This algorithm repeatedly solves a system of linear equations derived from the non-linear equations and their first derivatives until the global error is reduced below a threshold. The vertex positions from the current object are used as the initial solution for the new system of constraints. The solver tends to move the vertices only in small, well-controlled steps. As a result, solutions tend not to differ unnecessarily from the vertex geometry in the current object.

Once an acceptable surface topology and vertex geometry have been found, *Viking* replaces the current object description with the new interpretation. A new line drawing is then generated from the new current object description and the current view transform. The user can manipulate the new line drawing just like the old one, letting the user continue the cycle of modification and interpretation.

2.1.3 *Viking's* command modes

The four items shown in the center window of Figure 2-1 (*Edit, Move, Constraint* and *Component*) correspond to the four most commonly used modes in *Viking*. These modes determine how mouse actions in the line drawing's window are interpreted. If the user enters either *Constraint* or *Component* modes, the center window is overwritten with a specialized menu.

Edit mode is used for changing the appearance of the line drawing displayed in the image window. The user can draw new edges, delete existing edges and change the visibility of individual line segments while in this mode. Both the line drawing and the underlying object description change as a result of either of the first two actions. The last action, changing the visibility of a line segment, modifies only the line drawing, making it inconsistent with the underlying object description. The consistency between these two data-structures is restored by finding an interpretation of the line drawing. This is done automatically if the Autosolve switch is set.

Move mode is used for placing tacks, and moving vertices and edges. Tacks are constraints that either lock a vertex into a fixed position or force an edge to pass through a fixed point in space. If the Autosolve switch is set, *Viking* will use the constraint solver to maintain the constraints as the user drags a vertex or edge around with the mouse. Otherwise, the vertex or edge will follow the mouse without maintaining the constraints.

Constraint mode is used for placing or editing geometric constraints on the object. The constraint menu lets the user select a constraint template and then define constraints by picking vertices or edges to "fill in" the blanks. For example, the user can constrain two edges to have the same length by selecting the "equal length" template and then picking the edges to constrain. The user can also modify or delete previously defined constraints. Whenever the user adds a constraint, *Viking* will attempt to find a solution to the new system if the Autosolve switch is turned on.

Component mode is used for manipulating groups of vertices. A component contains a set of vertices and a transformation that maps the positions of these vertices from the local component

space to the global object space. This coordinate transformation is generated from eleven variables that control a component's size (using both an axis-independent variable and three axis-dependent variables), position, and orientation (using quaternions [25]). The user can lock or free these variables independently and the constraint solver can manipulate the free variables as needed when solving for a vertex geometry.

2.2 Examples

2.2.1 Creating a chair

This section describes a session using *Viking* to create an "easy chair." This example is somewhat contrived (for example, chairs are not normally made from homogeneous blocks) but it does convey the flavor of *Viking's* user interface. It also demonstrates how modifying the line drawing can be a substitute for using constructive solid geometry. It took me less than two minutes to transform the cube in Figure 2-4a into the chair in Figure 2-4i.

Preferred directions (see Section 2.1.1) were on automatic throughout this example. As a result, whenever the user started to draw an edge, *Viking* defined a set of context dependent vectors that could be used to position the edge's endpoint in three dimensions. For example, in Figure 2-4b, the new edge was projected onto a preferred direction that was parallel to the cube's rightmost vertical edge.

Figure 2-4a shows the initial object: a cube loaded from a library of standard objects. The first step in turning this cube into a chair is to add a raised back. Figure 2-4b shows the user drawing a new edge up from the upper-right corner of the cube. The user has finished drawing the edges for the chair's back in Figure 2-4c and is in the process of hiding the line-segments that would be obscured if the chair's back was solid and opaque.

In Figure 2-4d, the user deleted one unwanted vertex and is in the process of deleting the other (the user must pick a vertex twice to delete it: the first pick highlights the selected vertex, the second deletes it). These vertices are unwanted because deleting them and redrawing the missing edges ensures that the chair's back is a single, planar surface. If these vertices had not been deleted, *Viking* would have found an interpretation in which the chair's back and sides were each formed by two faces.

Deleting a vertex also deletes its adjacent edges and faces, although *Viking* preserves the hidden status of line-segments whose obscuring face is deleted. For example, in Figure 2-4d, the edge at the bottom-back of the cube is drawn with a single, thin line (indicating that it is adjacent to only one face) since the top, back and right faces of the cube were deleted when the first vertex was deleted. Also, the entire edge remains hidden, even though the face obscuring its right segment has been deleted.

Figure 2-4e shows the user redrawing some of the edges that were deleted when the user deleted the unwanted vertices. Figure 2-4f shows, from a different viewpoint, the user starting to draw a lowered seat on the first interpretation found for Figure 2-4e. This interpretation corresponds to a solid object. By default, *Viking* searches for "solid" interpretations and, as a result, may add faces that are not needed to generate an interpretation that is consistent with the line drawing. For example, the interpretation in Figure 2-4f contains a face on the front of the chair's back. An

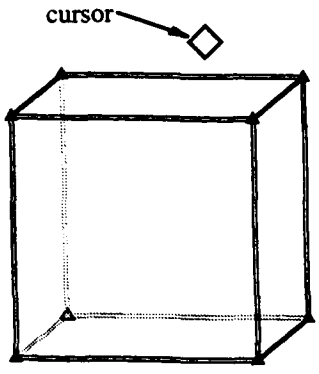


Figure 2-4a: Initial object: a unit cube.

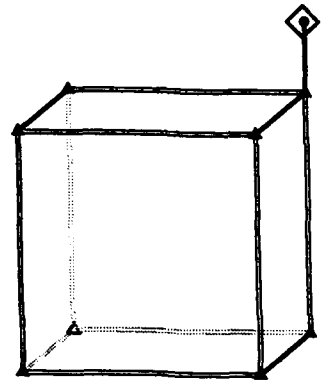


Figure 2-4b: Drawing the chair's back.

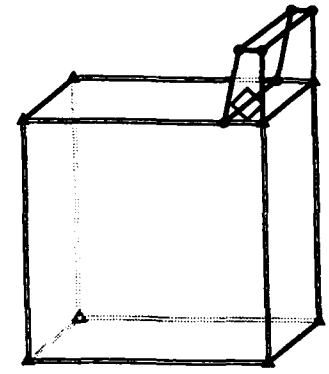


Figure 2-4c: Hiding obscured line-segments.

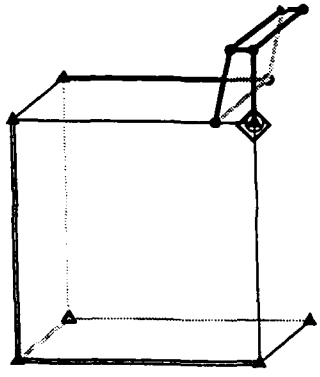


Figure 2-4d: Remove unwanted vertices and edges.

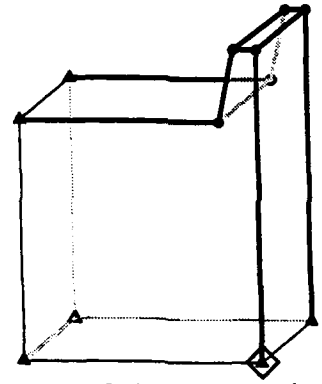


Figure 2-4e: Redraw the missing edges.

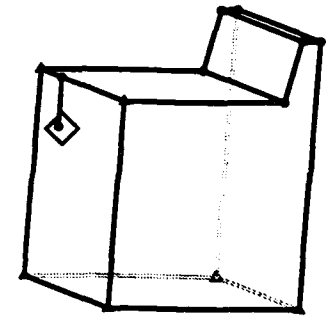


Figure 2-4f: Drawing the chair's seat.

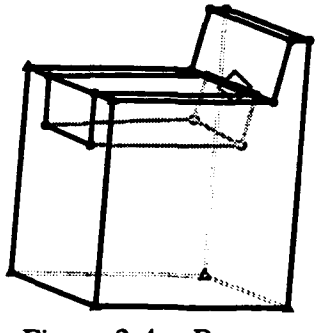


Figure 2-4g: Remove unwanted edges.

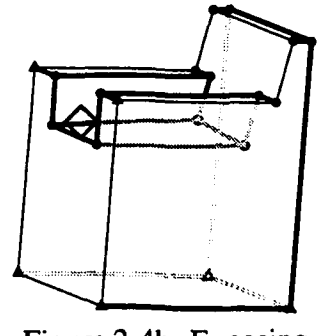


Figure 2-4h: Exposing visible line-segments.

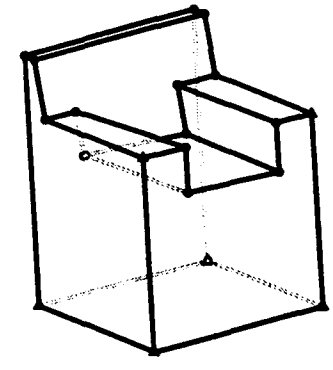


Figure 2-4i: The "completed" chair.

interpretation that did not contain this face, but was identical to the one in Figure 2-4f otherwise, would also be consistent with Figure 2-4e.

The user has finished drawing a lowered seat for the chair in Figure 2-4g and is in the process of removing some unwanted edges. In Figure 2-4h, the user is exposing the line-segments that would be visible if the chair's seat was lower than its arm rests. Figure 2-4i shows, from a different viewpoint, the first interpretation found for Figure 2-4h.

Even though the chair looks correct in Figure 2-4i, the geometry is not correct. For example, some edges that should be parallel to each other are skewed about 10°. These problems can be fixed in a minute or two by using geometric constraints. But, since the next example demonstrates the constraint solver, that part of the design process is skipped in this example.

2.2.2 An exercise in geometry

Suppose you have the following problem: if you place a solid equilateral tetrahedron face to face with a solid equilateral octahedron, how many faces does the resulting polyhedron have? The polyhedra are positioned and sized so that three of the tetrahedron's vertices coincide with three of the octahedron's vertices. Answering this question, by using *Viking* to create the object shown in Figure 2-5l, takes me less than three minutes.

Figure 2-5a shows the user starting to draw the two polyhedra. In Figure 2-5b, the user has changed the view transform by rotating it about the horizontal axis and is in the process of completing the octahedron's wire-frame. Figure 2-5c shows the user hiding the line-segments at the "back" of the polyhedra. Figure 2-5d shows the first interpretation found after hiding the rest of the line-segments that should be obscured.

The edges in Figure 2-5d were drawn without using either preferred directions or a cutting plane to position the vertices in three dimensions. The user made no attempt to draw the edges so that they all had exactly the same length. Instead, geometric constraints will be used to turn these "rough sketches" into equilateral polyhedra.

Figure 2-5e shows the effect of adding and solving for equal length constraints on the tetrahedron's edges by selecting the "$| \; v_a v_b \; | = 1.0 \; | \; v_c v_d \; |$" constraint template from the constraint menu (see Appendix B) and then selecting a pair edges to constrain. Figure 2-5f shows the effect of placing a similar set of constraints on the octahedron. The bent lines and "A" symbols indicate that the tetrahedron's edges all have the same length. The bent lines and "B" symbols do the same for the octahedron's edges. Lengths "A" and "B" may correspond to different lengths. In both Figures 2-5e and 2-5f, the vertices have moved to accommodate the constraints. Figure 2-5g, in which display of the constraints has been turned off, shows the two polyhedra from a different direction.

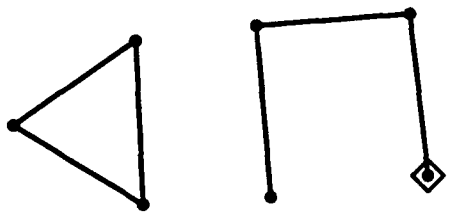


Figure 2-5a: Drawing the polyhedra.

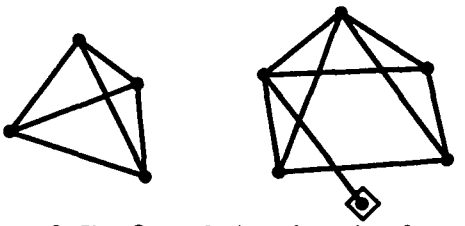


Figure 2-5b: Completing the wire-frames.

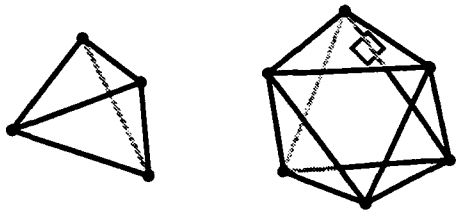


Figure 2-5c: Hiding obscured line-segments.

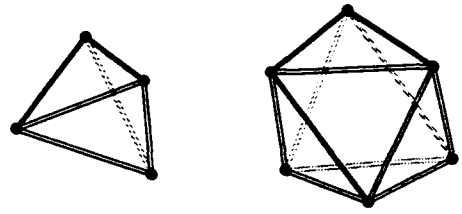


Figure 2-5d: Generating an interpretation.

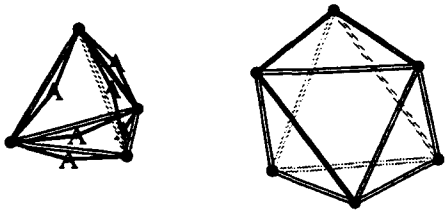


Figure 2-5e: Making an equilateral tetrahedron.

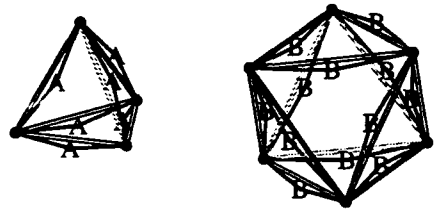


Figure 2-5f: Making an equilateral octahedron.

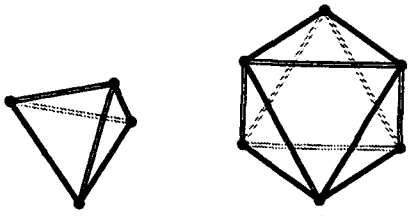


Figure 2-5g: Viewing from another direction.

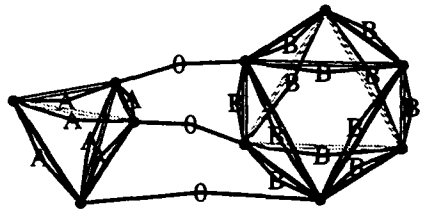


Figure 2-5h: Before solving the coincidence constraints.

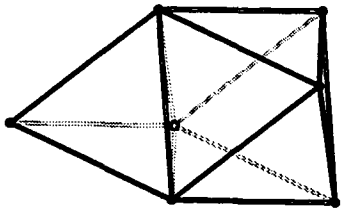


Figure 2-5i: After solving the coincidence constraints.

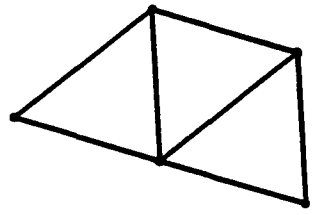


Figure 2-5j: An "edge-on" view.

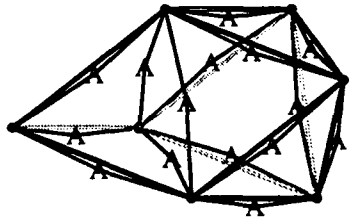


Figure 2-5k: The resulting seven-sided polyhedron.

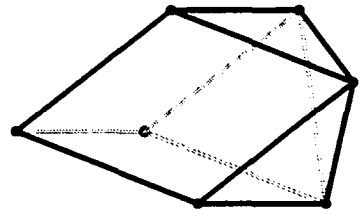


Figure 2-5l: Figure 2-5k with the constraints hidden.

In Figure 2-5h, the user has added, but not yet solved for, constraints forcing three of the tetrahedron's vertices to be coincident with three of the octahedron's vertices. The bent lines and "0" symbols indicate that the distance between the vertex pairs should be zero. Figure 2-5i shows the solution found by the constraint solver to the system described in Figure 2-5h. Figures 2-5h and 2-5i have, despite appearances, identical surface topologies: the constraint solver moved the vertices without changing the underlying structure.

In Figure 2-5j, the view transform has been changed to give a view "straight-down" one of the edges where the tetrahedron and octahedron are in contact. This view suggests that the vertices to either side of this edge are co-planar with the edge, forming a single four-sided face. In Figure 2-5k, the user has merged the six coincident vertices into three vertices, deleted the unwanted edges, and generated a new, seven-sided, interpretation. Figure 2-5l shows Figure 2-5k with all constraints hidden. *Viking* since it implicitly generates constraints that all faces are planar polygons, would not have been able to find a vertex geometry for Figure 2-5k unless the quadrilateral faces were planar polygons. The answer, therefore, to the question posed at the beginning of this section is that a tetrahedron and octahedron form a seven-sided polyhedron.

Chapter 3

Generating topologies from line drawings

Defining an object's surface topology – a list of an object's faces and the vertices that comprise each face – is an integral part of the design process. An object's surface topology can be used to calculate the object's mechanical properties (mass, moment of inertia, etc.). The surface topology can also be used to generate more "realistic" drawings on the object that help the user see and understand the object's shape and structure. This chapter describes *Viking's* algorithm for generating an object's surface topology from a line drawing.

3.1 The arc-labeling algorithm

Viking's users define an object's surface topology by modifying a line drawing of the object. Arc-labeling, an extension of Huffman-Clowes line-labeling [7, 17], generates surface topologies that are consistent with the modified line drawing. In the likely event that there is more than one consistent surface topology, the user can interactively search through these topologies to find the desired surface topology. A cost heuristic controls the order in which candidate surface topologies are generated: surface topologies that have the lowest cost are generated first. If the first surface topology found is not the one he or she wanted, then the user can guide the search by pointing out where the topology is "right" or "wrong."

Using arc-labeling to generate a surface topology from a line drawing is a three step process. First, all appropriate "labelings" are found for each of the object's vertices, where each labeling is a different configuration of surfaces adjacent to the vertex. Second, a branch and bound search generates mutually consistent labelings for the entire object in order of increasing cost. Third, a surface topology is found for each consistent labeling as it is generated.

The surface topology generated by this algorithm is displayed to the user, who can either accept or reject it. If the user rejects the surface topology, then steps two and three are repeated, possibly using feedback from the user to guide the branch and bound search. If the user accepts the surface topology, then the geometric constraint solver (see Chapter 4) finds a vertex geometry that satisfies the implicit and explicit constraints on the object. The surface topology and vertex geometry are then combined to create a new object description that replaces the current object description.

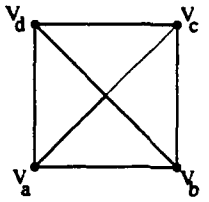


Figure 3-1: A line drawing that contains an occluding edge.

3.2 Finding the appropriate labelings for each vertex

The vertex labelings used by arc-labeling are a description of the surfaces adjacent to and in the immediate vicinity of a vertex. A valid labeling for a vertex is one that has a physical interpretation whose appearance is the same as the appearance of the vertex in the line drawing. An appropriate labeling is one that is both valid and consistent with other features in the line drawing, the user's preferences, and the appropriate labelings for the other vertices.

Viking uses an intersection library to determine the valid labelings for each vertex. Vertices with two, three or four adjacent edges are grouped into distinct categories so that all vertices in the same category have the same valid labelings. Chapter 5 describes the algorithm used to generate the valid labelings for an arbitrary vertex and the criteria used to group vertices. Appendix D contains *Viking's* intersection library.

3.2.1 Feature based restrictions

Some of the labelings that are valid for a vertex in general may be inappropriate in a particular line drawing. For example, in Figure 3-1, the edge between vertices V_b and V_d partially obscures edge $V_a V_c$. This situation can only be explained if one or more surfaces extend to the upper-right of edge $V_b V_d$, and no surfaces extend to its lower-left. Therefore, any labeling that violates this criterion is inappropriate even though it may be locally valid.

The rules used to filter out inappropriate labelings are summarized below:

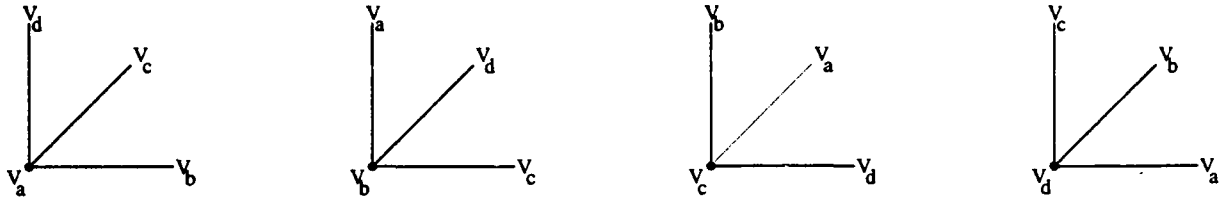
- Crossing-rule
  If two edges cross and one of the line-segments adjacent to the crossing is hidden, then the obscuring edge must have one or more adjacent surfaces extending over the obscured line-segment and no adjacent surfaces extending over the exposed line-segment.

  This rule assumes that it is possible to distinguish between intersections, where two or more edges meet at a vertex, and crossings, where two edges pass over one-another in the line drawing.

- Perimeter-rule
  A line-segment on the perimeter of a connected set of lines can only be adjacent to surfaces extending to the inside of the line drawing.

  This rule assumes that the entire object is contained within the boundaries of the line drawing and that none of the object's faces contain holes.

Labelings that violate one or both of these rules are eliminated from subsequent consideration.



Figures 3-2a through 3-2d: Extracting the vertices for labeling.

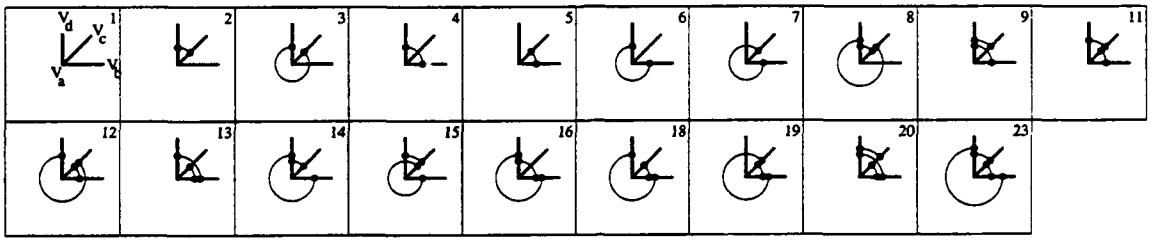


Figure 3-3a: Valid labelings for V_a.

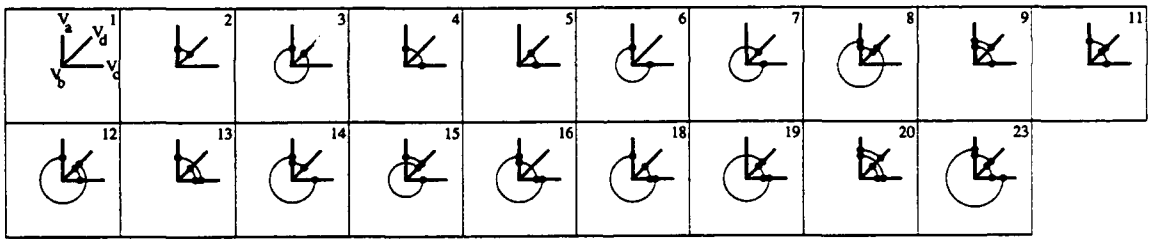


Figure 3-3b: Valid labelings for V_b.

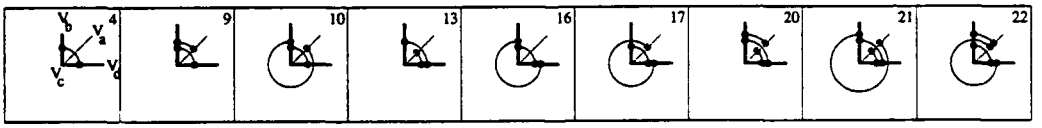


Figure 3-3c: Valid labelings for V_c.

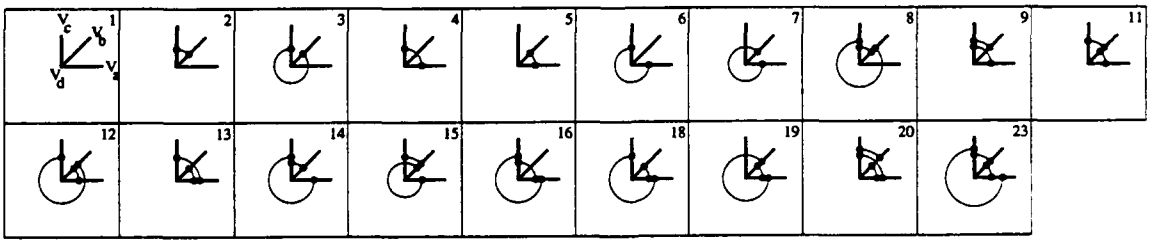


Figure 3-3d: Valid labelings for V_d.

3.2.2 User restrictions

Viking's users can specify how many faces should be adjacent to an edge when reviewing the surface topologies generated by arc-labeling. The user may either "sheath" or reject an edge. If the user sheaths an edge, then the number of surfaces adjacent to that edge will not change. If the user rejects an edge, then the number of surfaces adjacent to that edge will change. The restrictions imposed by sheathing an edge remain in effect until explicitly removed by the user. Those imposed by rejecting an edge only remain in effect until the user accepts an interpretation.

3.2.3 Mutual consistency based restrictions

An edge is consistently labeled if it has the same number of surfaces extending to each side according to the labeling selected for the vertices at either end. For example, edge V_aV_c is consistently labeled if both the labeling selected for V_a and the labeling selected for V_c has one surface extending to the lower-right of V_aV_c. A vertex's labeling is considered inappropriate if, as a result of using it to label the vertex, it becomes impossible to consistently label all the vertex's adjacent edges. For example, suppose one of the possible labelings for V_a has two surfaces extending to up from V_aV_b. If none of the labelings for V_b has two surfaces extending up from V_aV_b, then the labeling for V_a can be rejected as inappropriate.

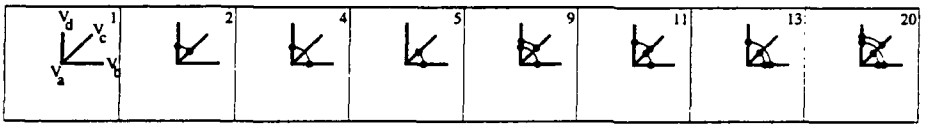
Waltz filtering [29] is an efficient algorithm for detecting and removing these locally inconsistent labelings. This algorithm tests each edge to see if the labelings for its vertices are mutually consistent, and deletes any inconsistent ones. Edges adjacent to a vertex that had one or more labelings deleted are then retested.

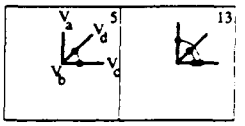
3.2.4 An example of finding the appropriate labelings

Figures 3-3a through 3-3d shows the valid labelings for the vertices in Figures 3-2a through 3-2d (see Chapter 5 and Appendix D). Note that the valid labelings for V_a, V_b, and V_d are identical, since these vertices all have the same intersection type (see Section 5.3.1). In these labelings, the lines represent the portions of an edge in the immediate vicinity of a vertex. The circular arcs correspond to surfaces between two edges around the vertex. For example, in labeling 2 for Figure 3-3a, vertex V_a is adjacent to one surface that extends left from edge V_aV_c to the right of edge V_aV_d. Note that edge V_aV_c is drawn with a solid line in Figure 3-3a and it is drawn with a dashed line in Figure 3-3c. This is because edge V_aV_c is visible near vertex V_a and hidden near vertex V_c. The first step in generating a consistent labeling for Figure 3-1 is to filter out all the inappropriate labelings for these vertices.

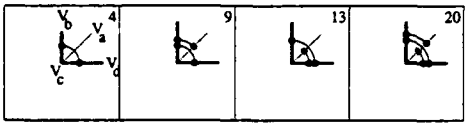
All the vertices in Figure 3-1 lie on the perimeter of the object. Therefore, according to the perimeter-rule (see Section 3.2.1), a labeling is inappropriate if it has any surfaces extending to the "outside" of the object. For vertices V_a, V_b and V_d, this eliminates labelings 3, 6-8, 12, 14-16, 18, 19 and 23. For vertex V_c, this eliminates labelings 10, 16, 17, 21 and 22.

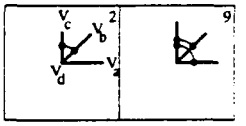
According to the crossing-rule (see Section 3.2.1), V_bV_d must have one or more surfaces extending to its upper-right and not have any surfaces extending to its lower-left. Therefore, labelings 1, 2, 4, 9, 11 and 20 are inappropriate for V_b and labelings 1, 4, 5, 11, 13 and 20 are inappropriate for V_d. Figures 3-4a through 3-4d show the vertices' remaining labelings after applying the feature based restrictions on the labelings.





Figures 3-4a and 3-4b: Appropriate labelings for V_a and V_b before applying the consistency rule.



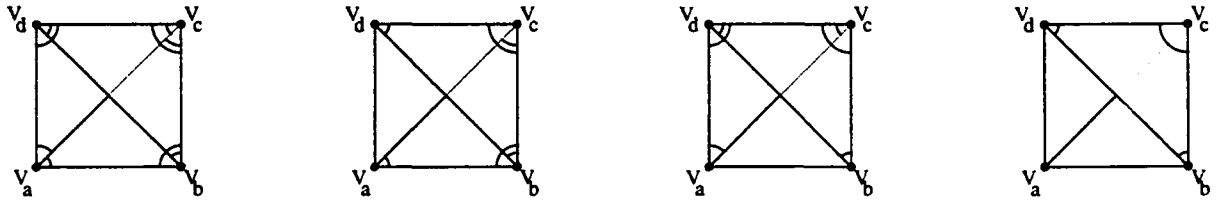


Figures 3-4c and 3-4d: Appropriate labelings for V_c and V_d before applying the consistency rule.

If the user has not sheathed or rejected any of the edges (see Section 3.2.2) in Figure 3-1, then only the mutual consistency restrictions are left to be applied. In labeling 13 of Figure 3-4a, V_aV_b has two surfaces extending to the inside of the object. None of the appropriate labelings for V_b, in Figure 3-4b, has two surfaces extending to the inside of the object from V_aV_b. Therefore, labeling 13 for V_a is locally inconsistent with V_b's labelings and can be removed. After comparing the labelings for V_a and V_d, a similar analysis can be used to delete labelings 9 and 20 from V_a's list of appropriate labelings.

3.3 Finding the best, consistent labeling

It is unusual for a line drawing to specify a surface topology uniquely. Normally, a line drawing will have several different interpretations. Figure 3-1, for example, has the four different interpretations shown in Figures 3-5a through 3-5d. Since the user is, typically, interested in only one of the many possible interpretations, some mechanism is needed so that the desired interpretation is found quickly and easily. In *Viking*, this is done by assigning a cost to each labeling and using a branch and bound search to generate labelings for the entire object in order of increasing cost. This heuristic seems to work well in practice: the first interpretation found by *Viking* is normally the interpretation that the user wanted.



Figures 3-5a through 3-5d: Interpretations generated for Figure 3-1.

3.3.1 Assigning costs to the labelings

A labeling's cost is based on several factors. The cost of a labeling is the sum of the costs for each of its adjacent edges. The edge cost is calculated as follows:

Cost	=	1	×	the change in the number of adjacent surfaces to either side of the edge +
		1	×	the number of adjacent surfaces that are different in the current object +
		9	×	the number of surfaces needed to satisfy the user's desired object type +
		36		if the edge has any adjacent surfaces and passes in front of another edge without obscuring it +
		72		if the edge's calculated visibility is different than its appearance.

The least important factor is how similar the surface topology described by the labeling is to the surface topology of the current object description: the more the labeling changes the surface topology, the greater the labeling's cost. This cost heuristic favors surface topologies that are similar to the current object's surface topology. It means that, normally, the object's surface topology only changes in the vicinity of a modification to the line drawing.

The next most important cost heuristic penalizes labelings that do not conform to the user's preferred object type. Currently, the user can indicate that he or she desires a "solid" or a "thin-shell" object. "Solid" objects are ones in which every edge is adjacent to exactly two faces. "Thin-shell" objects are ones in which every edge is adjacent to one or two faces. For every edge in a labeling that does not have the desired number of adjacent faces, the labeling's cost is increased appropriately. The user may also completely disable this heuristic.

The most important cost heuristic increases the cost of labelings that contain "implausible" edges. An edge is considered implausible if, using the vertex geometry from the current object, the calculated visibility of the edge differs from the appearance of the edge in the line drawing. For example, labeling 4 in Figure 3-4a has a surface between V_aV_b and V_aV_d that crosses V_aV_c. In order to be visible, V_aV_c must be in front of the crossing surface. If, according to the current object's vertex geometry, it is not, then V_aV_c is considered "implausible" and the cost of the labeling is increased.

3.3.2 Searching for a consistent labeling

Once a cost has been assigned to every labeling for every vertex, a branch and bound search is conducted to find the least expensive labeling that labels every edge consistently. This is done by creating a search tree in which every node or leaf in the tree corresponds to a partial labeling of the object. The search tree is initialized by creating a root node where no labelings have been assigned to any vertex in the object. Each iteration of the search cycle expands the leaf that has the lowest expected cost. A leaf is expanded by finding the unlabeled vertex with the smallest number of labelings that are consistent with the labelings selected for the leaf and its antecedents and creating a new leaf for each of these labelings. If the leaf being expanded has no unlabeled vertices, then the leaf is "expanded" by generating the surface topology corresponding to the labelings selected for each vertex and giving the user a chance to accept or reject it. If there is a tie for the leaf with the lowest expected cost, then the leaf with the smallest number of unlabeled vertices is selected.

A leaf's estimated cost is the sum of the costs of the labelings selected for the leaf and its antecedents and an estimate of the cost of labeling the unlabeled vertices. The cost estimate for the unlabeled vertices is found by totaling the costs the least expensive labeling for every unlabeled

	Labeling							
Vertex	1	2	4	5	9	11	13	20
V_a	54	36	108	36		18		
V_b				36			18	
V_c			36		18		18	0
V_d		36			18			

Table 3-1: Costs of the appropriate labelings in Figure 3-1.

vertex that is consistent with the leaf and its antecedents. Note that some of the labelings used to estimate the remaining cost may be mutually inconsistent, which is why this is an estimated rather than a true cost. This cost estimate is optimistic (i.e. the true cost will always be as high as or higher than the estimate). As a result, this algorithm is guaranteed to generate labelings in order of increasing total cost[1].

3.3.3 An example of searching for a consistent labeling

Suppose that, for purposes of this example, the costs in Table 3-1 are the costs assigned to the appropriate labelings for the vertices in Figure 3-1. These costs are similar to the costs that *Viking* would assign to the labelings. The only difference is that no penalty was assessed if the surface topology described by a labeling was unlike the surface topology of the current object. Note that the cost for V_a's labeling 4 is high because one of its edges is considered implausible (since it is assumed that V_c lies behind the plane defined by V_a, V_b and V_d).

Initially, the search tree contains a root in which none of the vertices are labeled. The estimated cost for the root node is 54 (18 + 18 + 0 + 18), meaning that the minimum cost to label the entire object will be at least 54. Since the root node is the only leaf, it is the leaf that is selected for expansion.

The first step in expanding a leaf is to decide which vertex to label. In this case, both V_b and V_d have two consistent labelings. Assume that V_b is chosen. This creates two new leaves, one in which V_b is assigned labeling 5 and one in which it is assigned labeling 13.

If labeling 5 is selected for V_b, then consistent labelings for the remaining unlabeled vertices are:

Vertex	Consistent labelings	Minimum cost
V_a	1 and 2	36
V_c	4 and 13	18
V_d	2 and 9	18

Since labeling 5 for V_b has a cost of 36, the expected cost for the new leaf is 108.

If labeling 13 is selected for V_b, then the consistent labelings for the remaining unlabeled vertices are:

[1]Proof: The cost of a terminal leaf – a leaf in which every vertex is labeled – is the true cost for labeling the line drawing. In order for a terminal leaf to be expanded, its cost must be as low as or lower than the cost of any other leaf in the tree. Since the cost of intermediate leaves are optimistic, the true cost of any other labeling must be greater than or equal to the cost of the terminal leaf.

Vertex	Consistent labelings	Minimum cost
V_a	4, 5 and 11	18
V_c	9 and 20	0
V_d	2 and 9	18

Since labeling 13 for V_b has a cost of 18, the expected cost for the new leaf is 54.

Therefore the leaf in which labeling 13 was selected for V_b has the lowest expected code and it is expanded next. Since V_c and V_d are tied for the lowest number of consistent labelings, assume V_c is labeled. This creates two new leaves, one in which V_c is assigned labeling 9 and one in which it is assigned labeling 20.

If labeling 9 is selected for V_c, then consistent labelings for the remaining unlabeled vertices are:

Vertex	Consistent labelings	Minimum cost
V_a	5	36
V_d	2	36

Since the total cost of the labelings selected for V_b and V_c is 36, the expected cost for the new leaf is 108.

If labeling 20 is selected for V_c, then consistent labelings for the remaining, unlabeled vertices are:

Vertex	Consistent labelings	Minimum cost
V_a	11	18
V_d	9	18

Since the total cost of the labelings selected for V_b and V_c is 18, the expected cost for the new leaf is 54.

Therefore the leaf in which labeling 20 was selected for V_c has the lowest expected code and it is expanded next. Since V_a and V_d have only one consistent labeling apiece, only one new leaf is created no matter which vertex is labeled. Since the expected cost of this leaf is 54, it is the leaf selected for the fourth expansion. This creates a new leaf with an estimated cost of 54. This leaf is then "expanded" by generating the surface topology corresponding to selecting, respectively, labelings 11, 13, 20 and 9 for V_a, V_b, V_c and V_d (see Figure 3-5a).

If the user rejected this surface topology, *Viking* would continue the branch and bound search. For this particular example, there are two leaves that can be expanded: the leaf in which labeling 5 was selected for V_b and the leaf in which labeling 9 was selected for V_c. Both leaves have the same estimated cost: 108. Since the costs are tied, the second leaf is selected because it has fewer unlabeled vertices. Therefore the next surface topology generated is the one that corresponds to selecting, respectively, labelings 5, 13, 9 and 2 for V_a, V_b, V_c and V_d (see Figure 3-5b).

3.4 Generating a surface topology

One a consistent labeling has been found, there is still a problem of generating the surface topology that corresponds to the labeling. A vertex's labeling only describes the surfaces in its immediate vicinity. Generating a surface topology is a matter of piecing the local surface

descriptions at each vertex together to form polygons. These polygons form the faces that comprise the object's surface topology.

The fragment of a surface described by a labeling can be represented as a list of three vertices: (V_a, V_b, V_c) describes a surface fragment adjacent to V_b that extends clockwise from V_aV_b around V_b to V_bV_c. Surface fragments can be joined "head-to-toe" to form cycles that eventually define one of the object's faces. Two surface fragments can be joined if the last two vertices in the first surface fragment are the same as the first two vertices in the second surface fragment. For example, using the labeling found for Figure 3-1 in Section 3.3.3, the labeling for V_a describes two surface fragments: (V_c, V_a, V_b) and (V_d, V_a, V_c). The labeling for V_b also describes two surface fragments: (V_d, V_b, V_c) and (V_a, V_b, V_c). The first surface fragment for V_a can be joined with the second surface fragment for V_b, forming a four vertex surface fragment: (V_c, V_a, V_b, V_c).

The process of joining surface fragments can be continued until a closed cycle is created. A surface fragment forms a closed cycle when the first two vertices in a surface fragment are equal to its last two vertices. For example, V_c is adjacent to three-surface fragments: (V_b, V_c, V_a), (V_b, V_c, V_d) and (V_a, V_c, V_d). Either the first or the second surface fragment can be joined to the four vertex surface fragment created earlier. If V_c's first surface fragment is joined, then a new, five vertex, surface fragment is created: $(V_c, V_a, V_b, V_c, V_a)$. This surface fragment describes a closed, three-sided, polygon that corresponds to one of the faces in this interpretation's surface topology.

If V_c's second surface fragment, instead of the first, had been joined to the four vertex surface fragment, then a different cycle would be created. If the "wrong" surface fragment is picked, then it is possible to generate an "illegitimate" surface topology. *Viking* considers a surface topology illegitimate if it contains a face in which either an edge or a vertex is repeated. In this particular case, picking V_c's second surface fragment would have led to a cycle in which V_c was repeated and could have led to a cycle that contained all nine surface fragments and defined a single, twisted face.

Viking's algorithm for generating a surface topology deals with this problem by using a heuristic to select the best surface fragment when there is a choice of surface fragments, and backtracking when the selected surface fragments leads to an illegitimate surface topology. The heuristic used to select surface fragments is a simple one: since faces are supposed to be planar, the surface fragment that lies closest to the plane defined by the "growing" surface fragment is selected first. The following algorithm, assuming that backtracking is implicitly supported, is used to generate a surface topology from a consistent labeling:

1. Make one surface fragment the "active" surface fragment.
2. Find the "best" surface fragment that can be joined to the active surface fragment.
3. Join the selected surface fragment to the active surface fragment.
4. Delete the selected surface fragment.
5. If the active surface fragment does not form a closed cycle, loop back to step 2.
6. Create the face that corresponds to the active surface fragment.
7. Delete the active surface fragment.
8. If there are any surface fragments left, loop back to step 1.

3.4.1 Automatically rejecting a surface topology

Although this algorithm will always generate a legitimate surface topology, it may not generate a "reasonable" one. Two additional checks are performed to detect and automatically reject unreasonable surface topologies. A surface topology is considered unreasonable if it contains either a bridging edge or two conflicting edges.

A bridging edge is one that extends across the inside of the face between two of the face's vertices. For example, in Figure 3-6a, edge V_iV_j would be a bridging edge if $V_eV_fV_jV_hV_gV_i$ formed a single face. This is because, if faces are planar polygons, then a bridging edge must lie in the plane defined by the face's vertices. Therefore, it is impossible for the geometric constraint solver to solve the implicit *in front of* or *behind* constraints that might be placed on the edge (see Section 4.1.2). It is normally possible to resolve the contradictions imposed by a bridging edge by using that edge to subdivide the "bridged" face. For example, in Figure 3-6a, the face defined by $V_eV_fV_jV_hV_gV_i$ could be replaced by two faces: $V_eV_fV_jV_i$ and $V_gV_iV_jV_h$.

Two edges conflict if they cross, all four line-segments adjacent to the crossing are visible, and both edges are adjacent to one or more surfaces. Conflicting edges are inconsistent with the line drawing because an edge with one or more adjacent surfaces must obscure any edge that crosses behind it. If both edges have one or more adjacent surfaces and neither edge is obscured, then both edges must cross behind the other, which is impossible.

3.4.2 Selecting the surface topology

Once a surface topology has been found, it is displayed to the user and he or she is given the following options:

- The user can accept it.
  Constraint satisfaction is then used to find the vertex geometry (see Chapter 4). Assuming a satisfactory vertex geometry is found, the new interpretation replaces the current object description.

  If a satisfactory vertex geometry is not found, then the user can either use the new surface topology with the old vertex geometry or continue the search for a surface topology.
- The user can reject it.
  Viking then continues the search for a consistent labeling (see Section 3.3.2).
- The user can sheath an edge (see Section 3.2.2).
- The user can reject an edge (see Section 3.2.2).
  This option also rejects the surface topology.

In practice, rejecting an edge seems to be the most effective mechanism of guiding the search for a desired topology. It is rarely needed, however, since the first surface topology found by *Viking* is normally the one that the user wanted.

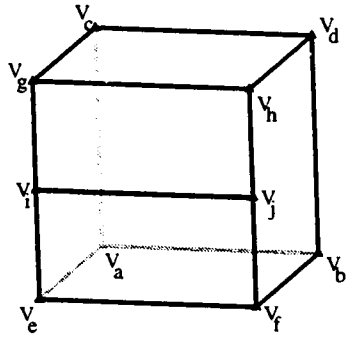


Figure 3-6a: A line drawing to be interpreted.

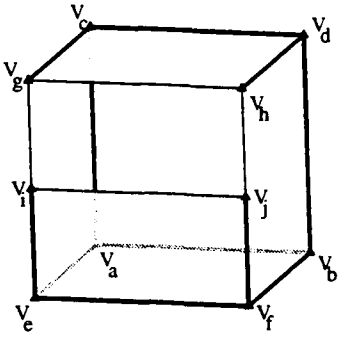


Figure 3-6b: An interpretation inconsistent with Figure 3-6a.

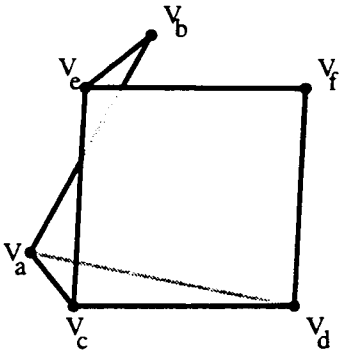


Figure 3-7: An edge with no consistent interpretation.

3.5 Problems

Most of the time, *Viking's* topology generation algorithm works correctly. There are, however, certain situations in which *Viking* either generates a surface topology that is inconsistent with the line drawing, or it cannot find the "desired" surface topology. Neither of these situations are common, but they can confuse the user when they occur.

3.5.1 Inconsistent interpretations

Figure 3-6a shows a line drawing in which the user has bisected the front face of a cube by drawing a new edge from one side to the other. Figure 3-6b shows one of *Viking's* interpretations of Figure 3-6a. This interpretation, however, is inconsistent with Figure 3-6a. In Figure 3-6a, the user has indicated that the middle line-segment of V_aV_c should be obscured. But, in Figure 3-6b, *Viking* has found an interpretation that contains an opening, bounded by V_g, V_i, V_j and V_h, though which it is possible to "see" the supposedly obscured line segment.

This problem occurred because none of *Viking's* rules for generating an interpretation required *Viking* to create a face bounded by $V_gV_iV_jV_h$. In this particular case, *Viking's* analysis of the crossing between edges V_aV_c and V_iV_j could not determine whether V_aV_c was hidden by surfaces extending to either side of V_iV_j, whether it was hidden by some other surface, or both. The same is true for *Viking's* analysis of the crossing between edges V_aV_c and V_gV_h. The interpretation shown in Figure 3-6b was generated on the incorrect assumption that there was some other surface that would hide the center line-segment of edge V_aV_c. Since no such surface existed, the interpretation was inconsistent with the line drawing that generated it.

3.5.2 Impossible interpretations

Another problem occurs when the user attempts to generate an interpretation of a line drawing like the one shown in Figure 3-7. In this line drawing, at V_c, there seems to be one surface that extends to either side of V_cV_e. But, at V_e, there seems to be one or more surfaces that extend to the right of V_cV_e. Unfortunately, this interpretation for V_cV_e violates the definition of a consistent

labeling for an edge (see Section 3.2.3). Therefore, in this particular case, *Viking* will not be able to generate any interpretation of the line drawing, much less the one desired by the user.

If all faces are planar, then this situation is rare since it can only occur if the vertices in a potential face are not co-planar and the view transform causes the potential face to fold back on itself as in Figure 3-7. This problem can normally be resolved by moving one or more vertices so that they all lie in the same plane or changing the view transform. If faces can be non-planar, however, then this situation may be more common since the vertices forming a face are less likely to lie in the same plane.

One way to avoid this problem is to relax the definition of a consistently labeled edge so that an edge is considered consistently labeled if it has the same number of adjacent surfaces at both ends, instead of the same number of surfaces extending to either side at both ends. Unfortunately, relaxing the definition increases the number of consistent labelings that can be found. This, in turn, may make it more difficult for the user to find the desired interpretation. A better solution may be to modify the definition of a consistently labeled edge so that non-planar faces can "switch sides," but planar faces can not.

Chapter 4

Solving systems of geometric constraints

Viking splits the process of generating a solid interpretation of line drawing into two parts: finding the surface topology and solving for a satisfactory vertex geometry. Chapter 3 described an algorithm for finding the surface topology. This chapter describes an algorithm for using that surface topology to solve for a satisfactory vertex geometry: one that satisfies the object's explicit and implicit geometric constraints. *Viking* solves these constraints by iteratively solving a linear programming problem generated from the non-linear equations that correspond to the constraints and their first derivatives.

The explicit constraints let the user precisely define the object's desired geometry. For example, the user can specify that an edge has a length of 2.72, that an angle spans 26.57° or that two edges have the same length. *Viking* combines these constraints with constraints derived from the line drawing and surface topology. The resulting constraint network is rewritten as a non-linear satisfaction problem and solved by "hill-climbing" from the current vertex geometry. As might be expected, *Viking* has trouble when the constraints are infeasible or specify a radically different vertex geometry. The user can alleviate the latter problem, however, by using direct manipulation to move the vertices to "about" the correct position.

4.1 Generating the implicit constraints

The implicit constraints help ensure that the new object description is consistent with the line drawing. There are three types of implicit constraints: world, image and dragging.

- World constraints force every face to be a planar polygon.
- Image constraints force hidden lines to pass behind obscuring faces and lines.
- Dragging constraints force a vertex to follow the mouse when the user is interactively moving a vertex.

These constraints, by themselves, cannot turn a 2D sketch into a reasonable 3D object. For example, applying these constraints to a "flattened" cube (an object that looks like a cube from one viewpoint but whose vertices all lie in the XY-plane) will generate a vertex geometry in which some vertices are slightly displaced in front of or behind the XY-plane. This geometry satisfies the implicit constraints but it is an interpretation that a reasonable user would neither expect nor want.

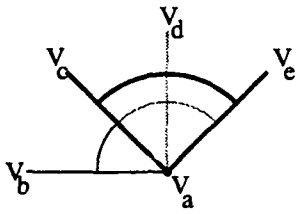


Figure 4-1: Multiple spanning surfaces.

4.1.1 World constraints

The world constraints force every face on the object to be a planar polygon. These constraints are the easiest type of constraint to generate. Since the object's surface topology is known, generating these constraints is simply a matter of constraining the vertices in each face to be co-planar with one another.

4.1.2 Image constraints

The image constraints help guarantee that an object looks "right" by forcing the hidden parts to be behind something and the visible parts to be in front of everything. Normally, these constraints only have an effect on the resulting vertex geometry when the surface topology has changed. Therefore, in order to improve performance, *Viking* only generates these constraints after finding a new surface topology (see Chapter 3).

There are two types of implicit image constraints:

- Crossing constraints: at any crossing where two lines cross one another without physically intersecting, the visible line must be in front of the hidden or partially hidden line, and
- Spanning constraints: at any visible vertex, adjacent, visible line-segments must be in front of any spanning surface fragments and adjacent, hidden line-segments must be behind at least one spanning surface fragment.

The spanning constraints are similar to the first two validity criteria given in Section 5.2.

A problem occurs in situations like that shown in Figure 4-1, where an adjacent, hidden line-segment that is spanned by two or more surface fragments. A line-segment will be hidden as long as it is behind one of the spanning surface fragments: it does not have to be behind all of them. This complicates constraint generation since *Viking* can only constrain a line-segment with respect to a single surface fragment: it can not create a constraint that says "V_aV_d is behind either (V_b, V_a, V_e) or (V_c, V_a, V_e)."

Viking solves this problem by determining which surface fragment must be in front of the others and constraining the line-segment to lie behind this surface fragment. In Figure 4-1, (V_c, V_a, V_e) must be in front of (V_b, V_a, V_e) since line-segment V_aV_c is visible. If V_aV_c were hidden, then (V_b, V_a, V_e) would be in front. It is always possible to determine the foremost surface fragment when a vertex has four adjacent edges, and the problem does not arise when a vertex is adjacent to three or fewer edges. Since *Viking* is limited to objects in which every vertex is adjacent to four or fewer edges, it is always able to determine the foremost surface fragment.

4.1.3 Dragging constraints

A common operation in *Viking* is when the user drags a vertex to a new position using the mouse while maintaining the constraints. *Viking* uses two different techniques to implement dragging, depending on whether the dragged vertex is a member of a component[1] or not. No matter which technique is used, however, the effect is the same: the vertex seems to follow the mouse while the rest of the object adjusts to maintain the constraints.

If the dragged vertex does not belong to a component, *Viking* does the following:

1. Set the weight of the dragged vertex to 10 (see Section 4.4.1).
2. Move the dragged vertex to the position indicated by the mouse.
3. Solve for a new vertex geometry.
4. If the a mouse button is still pressed, loop back to step 2.
5. Set the weight of the dragged vertex to 1.

Increasing the weight of the vertex makes it more difficult for the constraint solver to move the vertex away from its new position, aiding the illusion that the vertex is being dragged by the mouse. One problem with this algorithm is that solving for a new vertex geometry can take several seconds.

If the dragged vertex is a member of a component, then the algorithm given above will not work since it is not always obvious how to modify the component's transformation variables to move the dragged vertex to follow the mouse. The algorithm used in this case is the following:

1. Create a new vertex whose weight is 10.
2. Add a constraint that the distance between the new vertex and the dragged vertex is 0.
3. Move the new vertex to the position indicated by the mouse.
4. Solve for a new vertex geometry.
5. If the a mouse button is still pressed, loop back to step 3.
6. Delete the distance constraint and the new vertex.

The high weight of the new vertex "encourages" the constraint solver to find a solution in which the dragged vertex seems to follow the mouse.

4.2 Generating the explicit constraints

Explicit constraints are created by the user and stored in the current object description. *Viking's* users add constraints to the object by filling in constraint templates. For example, by selecting the "equal length" template and picking vertices v_a, v_b, v_c, and v_d the user can add a constraint that the distance between v_a and v_b is equal to the distance between v_c and v_d.

[1] A component is a distinct group of vertices within the object that has its own coordinate transform.

4.3 Representing constraints with equations

Every geometric constraint is represented by one or more non-linear equations (see Section C.1). These equations are, generally, functions of the vertices' object space coordinates. Combining these equations defines a non-linear satisfaction problem that has following form:

Solve for $\vec{x}$ such that:

$$\begin{aligned} h_i(\vec{x}) &= 0 \quad (0 \leq i \leq \mid \vec{h} \mid) \\ g_j(\vec{x}) &\geq 0 \quad (0 \leq j \leq \mid \vec{g} \mid) \end{aligned}$$

The coordinates of a locked vertex are treated as fixed constants. The coordinates of a free vertex are independent variables that can be modified by the constraint solver.

Planarity constraints are unique in that each planarity constraint generates four new variables that define a common plane for the constraint's vertices. Another unique aspect of planarity constraints is that they are placed on either the object space coordinates or the image space coordinates of the vertices. In the latter case, the new variables describe a plane in image space that contains the vertices. This plane description is used by the spanning constraints to determine whether a line-segment is in front of or behind a surface fragment. In either case, the new variables are automatically locked if the constraint's locked vertices describe a well defined plane. If the new variables are not locked, then they can be manipulated by the constraint solver in the same way that the coordinates of a vertex are manipulated.

4.3.1 Pseudo-variables

Viking's constraint equations (see Section C.1) can be functions of true variables, which are traditional free variables, pseudo-variables, which are functions of other variables, or both. For example: a vertex's image space coordinates are represented by three pseudo-variables that depend on the vertex's object space coordinates (which are normally true variables) and the view transform (see Section C.2). The image space coordinates are used by the image constraints since this is the most convenient way to express "in front of" or "behind" constraints.

When a vertex is added to a component, its object space coordinates are then represented by pseudo-variables that depend on the vertex's component space coordinates and the component transform. Modifying the component's transform is, therefore, equivalent to modifying the object space coordinates of every vertex in the component. The component transform has four parts (see Section C.3).

- u The uniform scaling parameter (1 variable).
- $\vec{\sigma}$ The axis-dependent scaling vector (3 variables).
- $\vec{\delta}$ The translation vector (3 variables).
- $\vec{q}$ The orientation quaternion (4 variables) [25].

Each of these parts can be independently locked. For example, if the user locks a component's scaling variables, then the component will move as if it were a rigid body.

4.3.2 Discarding redundant constraints

In general, analyzing geometric constraints for redundancy is difficult. *Viking*, however, is able to detect certain classes of redundant constraints on components. For example, if the component is only free to move as a rigid body, then any angular or distance constraint involving only the component's vertices are redundant. The following rules are used to discard redundant constraints if all of a constraint's vertices belong to the same component:

- Planarity constraints are always considered redundant.
- Angular constraints are considered redundant if the component cannot scale independently along each axis ($\vec{\sigma}$ is locked).
- Distance constraints are considered redundant if the component can not scale either uniformly or independently along each axis (u and $\vec{\sigma}$ are locked).

4.4 Solving non-linear equations

Viking solves systems of non-linear equations by splitting them into their independent components using the following algorithm to solve each component:

1. Finding a displacement by creating and solving a linear optimization problem.
2. Finding an approximation to the optimal displacement by using a binary search.
3. Adding this displacement to the variables.
4. If some of the equations are unsatisfied and satisfactory progress is being made, loop back to step 1.

4.4.1 Finding the displacement

The goal of each iteration cycle is to reduce the global error in the system of non-linear equations. The first step in this process is to find a displacement in which the global error decreases. This is done by creating and solving the following linear optimization problem:

$$\text{Minimize: } \sum_{i=1}^{|\vec{h}|} \mid e_i \mid + \sum_{j=1}^{|\vec{g}|} s_j + v \sum_{k=1}^{n} (w_k \cdot \mid d_k \mid)$$

$$\begin{array}{lrcll} \text{Such that:} & h_i(\vec{x}) + \nabla h_i(\vec{x}) \cdot \vec{d} + e_i & = & 0 & 0 \le i < \mid \vec{h} \mid \\ & g_j(\vec{x}) + \nabla g_j(\vec{x}) \cdot \vec{d} + s_j & \ge & 0 & 0 \le j < \mid \vec{g} \mid \\ & s_j & \ge & 0 & 0 \le j < \mid \vec{g} \mid \\ & \mid d_k \mid & \le & (b/w_k) & 0 \le k < n \end{array}$$

$$\text{Solve for: } \vec{d}, \vec{e}, \vec{s}$$

Where:

- n is the number of variables.

- $| \vec{h} |$ is the number of non-linear equations of the form: $h_i(\vec{x}) = 0$.
- $| \vec{g} |$ is the number of non-linear equations of the form: $g_j(\vec{x}) \geq 0$.
- $\vec{x}$ is the initial position vector.
- $\vec{d}$ is the displacement vector.
- $\vec{e}$ and $\vec{s}$ are the remaining error variables.
- w_k is the weight for variable k (typically $1 \leq w_k \leq 10$).
- v is the displacement bias (currently 10^{-3}).
- b is the maximum step size (currently 5% the length of the diagonal of the object's bounding box).

This linear programming problem is similar to the one developed by Bullard and Biegler [4] to solve systems of non-linear equations[2]. This problem is not in standard form, since the absolute value function is used in the objective function. It is, however, easy to convert this problem to standard form by:

- replacing e_i with $q_i - r_i$,
- replacing $| e_i |$ with $q_i + r_i$, and
- adding the equations $q_i \geq 0$ and $r_i \geq 0$.

A similar substitution is used for $\vec{d}$.

Since the displacement is found using a linear approximation of a non-linear system, it may not be optimal for reducing the global error. Indeed, for some systems, the global error will increase if the vertices are moved by the displacement. *Viking* uses two techniques to mitigate this problem. First, because the linear approximations are more accurate for small displacements, the objective function favors solutions with smaller displacements. Second, *Viking* searches along the displacement vector to find the distance that produces the smallest global error.

4.4.2 Finding an approximation to the optimal displacement

A binary search along the vector defined by the displacement is used to find the displacement with the smallest global error. The displacement found by the search is only an approximation of the optimal displacement, since the search is limited to sampling a small number of points along a single vector that may not contain the true optimal displacement. The algorithm used to perform

[2]The only significant differences are that, in Bullard and Biegler's version, $v = 0$ and a different strategy is used to find the approximation to the optimal displacement.

the binary search is given below (where $\mu(\vec{x})$ is the global error at $\vec{x}$):

Starting with:	$l \Leftarrow 0.1$	[1]
	$h \Leftarrow 1$	[2]
Loop at most 5 times:		[3]
	$m \Leftarrow (l+h)/2$	
	if $((\mu(\vec{x}+h\vec{d}) < \mu(\vec{x}+l\vec{d})) \wedge (\mu(\vec{x}+h\vec{d}) < \mu(\vec{x}+m\vec{d})))$	[4]
	exit loop.	
	else if $(\mu(\vec{x}+l\vec{d}) < \mu(\vec{x}+h\vec{d}))$	
	$h \Leftarrow m$	[5]
	else	
	$l \Leftarrow m$	[6]

$(h\vec{d})$ is the displacement used to move the vertices.

The parameters for this search were determined through trial and error as *Viking* was being written. The initial values for h and l (lines [1] and [2]) were selected because they seemed to give the best performance. Different values for h, for example, seemed to make the system either slower or less predictable. Setting l to zero seemed to make it more likely that the system would become trapped in a local minima. The same is true for the maximum number of times the loop was allowed to execute (line [3]). When less than five loop iterations were allowed, the algorithm seemed to "fail" more frequently than it did when five (or more) loop iterations were allowed. Allowing more than five iterations, however, seemed to increase the time required to find a solution without noticeably decreasing the number of failures.

The benefit of using the binary search is most clearly seen when $\vec{x}$ is close to a solution for the system of equations. In this situation, the initial displacement may be too large. Without the binary search, the vertices would be displaced beyond the solution. On the next iteration, the displacement may overshoot the solution again and the constraint solver will oscillate around the solution without finding it. Using the binary search described above mitigates this problem. If the displacement steps too far beyond the solution, then the global error will not decrease, the test on line [4] will be false, and the search will continue on either the first half of the displacement vector (line [5]) or the second half (line [6]). In most cases (such as while solving the systems of equations generated while creating the examples in Chapter 2), the search loop ended after two or three iterations.

The global error is the sum over all equations of each equation's error function (see Section C.1). The error functions are similar to the constraint functions used to construct the LP (see Sections 4.4.1 and C.1). The primary difference is that the constraint functions are, as much as possible, designed to be have small second (and higher) derivatives and the error functions are designed to measure the error in units that are roughly proportional to the amount the variables need to change to eliminate the error. In general, the error function is simply a permutation of the constraint function raised to some power (see the formulation of the angular constraint in Section C.1 for an example of this).

4.4.3 Adding the displacement to the variables

Once a displacement has been found, it is added to the values of the variables. A check is then performed to see if satisfactory progress is being made. *Viking* currently bases progress on the slope of the change in global error over the past fifteen iterations. If this slope is close to zero or positive (which would indicate an increasing global error), the search ends even though some of the constraints are left unsatisfied.

4.5 Solving non-linear equations and other black magic

Solving systems of non-linear equations is, at best, chancy business since there are no algorithms that are guaranteed either to find a solution or to determine that no solution exists. All of the algorithms for simultaneously solving systems of non-linear equations, therefore, tend to be of the form: "beat the numbers with a stick until either the numbers break or the stick does." As might be expected, these algorithms are very sensitive to the nature of the equations they are solving and the initial conditions. Change either and an algorithm that worked reliable may fail miserably.

This chapter describes the algorithm *Viking* uses to solve systems of non-linear equations. It is the result of combining a theory (based on Bullard and Biegler's algorithm [4]) with a lot of tinkering to make it work. Unfortunately, the tinkering process can continue far beyond the point of diminishing returns. I did not, for example, try every possible formulation of the constraint and error functions. Instead, I found something that seemed to work most of the time and went on to other things. In the cases where I did a lot of tinkering (such as the parameters for the search loop in Section 4.4.2), I have tried to indicate what some of the trade-offs were. In other cases, the first thing I tried seemed to work (such as the maximum step size in Section 4.4.1) and I do not know what the effect of changing it would be.

Chapter 5

Finding a vertex's valid labels

The algorithm described in Chapter 3 for generating a surface topology from a line drawing used an intersection library (see Section 3.2 and Appendix D) to find the valid labelings: a labeling was considered valid if and only if it existed in the intersection library. This chapter describes arc-labeling, which is an algorithm to test the validity of a labeling for arbitrary vertices by determining whether it has a physical interpretation whose appearance matches the appearance of the vertex in the line drawing.

Viking's intersection library was created by using arc-labeling to test the validity of every possible labeling for each distinct vertex types. Arc-labeling, however, is not limited to testing the validity of labelings for the vertex types in the intersection library. Instead, arc-labeling can be used with vertices adjacent to any number of edges and labelings containing any number of surfaces adjacent to each edge.

5.1 Line-labeling vs. arc-labeling

In arc-labeling, an edge's label tells how many surfaces extend to either side of the edge and indicates the left and right bounding edges for each surface. The surface can be thought of as a solid arc extending left from one edge around the vertex to the right of another edge. Huffman-Clowes line-labeling [7, 17], in contrast, uses four different edge labels: convex (where the edge lies along a ridge formed by the two surfaces), concave (where the edge lies at the bottom of a valley formed by the two surfaces), and left or right occluding (where both surfaces extend to the left or right of the edge). The primary difference between the two systems is that arc-labeling can represent edges adjacent to any number of surfaces and that line-labeling can distinguish between convex and concave arrangements of the adjacent surfaces.

Another difference between the two systems is that line-labeling algorithms typically use a pre-computed intersection "library" (also called a junction library) which contains all valid labelings for each distinct vertex type. The intersection library is normally hand generated and only contains vertices with two or three adjacent edges, though there are some exceptions [18]. Arc-labeling can be used to test the validity of a labeling for any vertex, though the time required to test a labeling grows exponentially with the number of adjacent edges. In arc-labeling, as with traditional line-labeling, it is possible to group vertices into distinct types and pre-compute the valid labelings for each type. Appendix D shows *Viking's* intersection library, which contains the valid labelings for

any vertex with two, three or four adjacent edges.

Both line-labeling and arc-labeling assume that the line drawing represents a general view of the object [24]. In essence, this is an assumption that a small change in the observer's view point will produce a correspondingly small change in the line drawing. This assumption has three corollaries in *Viking*:

1. Every face is drawn as a closed polygon with a non-zero area.
2. Every edge is drawn as a line with a non-zero length.
3. If two adjacent lines are parallel, then the corresponding edges are parallel.

These corollaries eliminate several special cases (in particular, what is left and right for a line with a length of zero?), but make it impossible to use *Viking* to analyze many engineering drawings since these drawings often violate one or more of the corollaries.

5.2 Testing a labeling for validity

A model of a vertex's labeling can be created by assigning a position to the endpoint of each adjacent edge. Using this model, the visibility of each adjacent edge can be calculated and the surface fragments can be tested to see if they intersect anywhere except along a common boundary. If the endpoints can be positioned such that the appearance of the model is the same as the appearance of the vertex in the line drawing and none of the surface fragments intersect one another, then a physical model of the labeling exists and labeling is valid.

It is possible to write first order logic expression that corresponds to the criteria above using the notation in Table 5-1. These expressions (along with their English translations) are as follows:

- Every visible edge is in front of every spanning surface fragment:
  $$(\forall v \in V, (\forall \mathcal{X} \in S \mid (v \propto \mathcal{X}), (v \Uparrow \mathcal{X}))) \tag{5.1}$$
- Either the entire vertex is hidden by a non-adjacent surface or
  every hidden edge is behind at least one spanning surface fragment:
  $$((V = \emptyset) \vee (\forall h \in H, (\exists \mathcal{X} \in S \mid (h \propto \mathcal{X}), (h \Downarrow \mathcal{X})))) \tag{5.2}$$
- No two distinct surface fragments intersect anywhere except along a bounding edge:
  $$(\forall \mathcal{X} \in S, (\forall \mathcal{Y} \in S \mid (\mathcal{Y} \neq \mathcal{X}), (\mathcal{X} \otimes \mathcal{Y}))) \tag{5.3}$$

A vertex labeling is considered valid if and only if it is possible to create a model of the labeling whose appearance matches that of the vertex in the line drawing. This is possible if and only if all three of these expressions can be simultaneously satisfied.

Using the other predicates using the expansion given in Table 5-2, the $\otimes$ predicate can be written as a more complicated expression using the other predicates. Table 5-2 was generated by enumerating all valid arrangements of two surface fragments around a point (see Figures 5-1a through 5-9). The other predicates, except for the $\Uparrow$ and $\Downarrow$ can be evaluated based on the angle each edge makes with the horizontal. Therefore, the validity criteria can be reduced to a Boolean expression containing only $\Uparrow$ and $\Downarrow$ terms.

The algorithm for determining the validity of an arbitrary labeling is as follows:

$a, b, \ldots$ = edges

$\mathcal{X}, \mathcal{Y}, \ldots$ = surface fragments

V = Set of all visible edges adjacent to the vertex.

H = Set of all hidden edges adjacent to the vertex.

S = Set of all surface fragments adjacent to the vertex.

$\mathrm{l}_{\mathcal{X}}$ = The edge forming the left boundary of $\mathcal{X}$.

$\mathrm{r}_{\mathcal{X}}$ = The edge forming the right boundary of $\mathcal{X}$.

$$(a \propto \mathcal{X}) = \begin{cases} \text{True} & \text{If } \mathcal{X} \text{ spans } a \ (a \text{ lies between the bounding edges of } \mathcal{X}, \\ & a \neq \mathrm{l}_{\mathcal{X}} \text{ and } a \neq \mathrm{r}_{\mathcal{X}}). \\ \text{False} & \text{otherwise} \end{cases}$$

$$(a \Uparrow \mathcal{X}) = \begin{cases} \text{True} & \text{If } a \text{ is in front of the plane defined by } \mathcal{X}. \\ \text{False} & \text{otherwise} \end{cases}$$

$$(a \Downarrow \mathcal{X}) = \begin{cases} \text{True} & \text{If } a \text{ is behind the plane defined by } \mathcal{X}. \\ \text{False} & \text{otherwise} \end{cases}$$

$$(\mathcal{X} \otimes \mathcal{Y}) = \begin{cases} \text{True} & \mathcal{X} \text{ and } \mathcal{Y} \text{ do not intersect except along a bounding edge.} \\ \text{False} & \text{otherwise} \end{cases}$$

Table 5-1: Notation key for validity expressions.

1. Generate Boolean expressions corresponding to expressions (5.1) – (5.3), based on the vertex and the labeling.
2. Combine the three Boolean expressions into a single expression (ANDing the three clauses together).
3. Simplify the Boolean expression by:
   - rewriting the $\otimes$ predicate expressions using the other predicates, and
   - evaluating all except the $\Uparrow$ and $\Downarrow$ predicates.
4. Rewrite the Boolean expression in disjunctive normal form. This expression will have the form:

$$B = C_1 \vee C_2 \vee \cdots \vee C_n \tag{5.4}$$

   where each C_i is an expression of the form:

$$(a_1 \Uparrow \mathcal{X}_1) \wedge (a_2 \Uparrow \mathcal{X}_2) \wedge \cdots \wedge (a_n \Uparrow \mathcal{X}_n) \wedge (a_{n+1} \Downarrow \mathcal{X}_{n+1}) \wedge \cdots \wedge (a_m \Downarrow \mathcal{X}_m)$$

5. Use the algorithm given in Section 5.3 to determine the satisfiability of each C_i in expression (5.4). If one or more terms are satisfiable, the expression as a whole is satisfiable. This means that the labeling satisfies the validity criteria and, therefore, has a physical interpretation whose appearance matches the appearance of the vertex in the line drawing.

$$
\begin{array}{rcll}
\angle a & = & \text{The angle } a \text{ makes with the horizontal.} & \\
\angle ab & = & \text{The angle counterclockwise from } a \text{ to } b. & \\
\angle \mathcal{X} & = & \text{The angle between the bounding edges of } \mathcal{X}. & \\
(a \not\propto \mathcal{X}) & = & \neg(a \propto \mathcal{X}) & \\
 & & & \\
(\mathcal{X} \otimes \mathcal{Y}) & = & \eta(\mathcal{X},\mathcal{Y}) \vee \kappa(\mathcal{X},\mathcal{Y}) \vee \varrho(\mathcal{X},\mathcal{Y}) \vee \tau(\mathcal{X},\mathcal{Y}) \vee \psi(\mathcal{X},\mathcal{Y}) \vee \omega(\mathcal{X},\mathcal{Y}) \vee & \\
 & & \eta(\mathcal{Y},\mathcal{X}) \vee \kappa(\mathcal{Y},\mathcal{X}) \vee \varrho(\mathcal{Y},\mathcal{X}) \vee \tau(\mathcal{Y},\mathcal{X}) \vee \psi(\mathcal{Y},\mathcal{X}) \vee \omega(\mathcal{Y},\mathcal{X}) & \\
 & & & \\
\eta(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \propto \mathcal{X}) \wedge (\eta_1(\mathcal{X},\mathcal{Y}) \vee \eta_2(\mathcal{X},\mathcal{Y})) & \\
\eta_1(\mathcal{X},\mathcal{Y}) & = & (\angle\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Downarrow \mathcal{X})) \vee & \text{[Figure 5-1a]} \\
 & & ((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Uparrow \mathcal{X})) \vee & \text{[Figure 5-1b]} \\
 & & ((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Uparrow \mathcal{X})) \vee & \text{[Figure 5-1c]} \\
 & & ((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Downarrow \mathcal{X}))) & \text{[Figure 5-1d]} \\
\eta_2(\mathcal{X},\mathcal{Y}) & = & (\angle \mathrm{r}_\mathcal{X}\mathrm{l}_\mathcal{Y} < 180°) \wedge (\angle \mathrm{r}_\mathcal{Y}\mathrm{l}_\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Downarrow \mathcal{X})) \vee & \text{[Figure 5-2a]} \\
 & & ((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \Uparrow \mathcal{X}))) & \text{[Figure 5-2b]} \\
 & & & \\
\kappa(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\angle\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Uparrow \mathcal{Y})) \vee & \text{[Figure 5-3a]} \\
 & & ((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \Downarrow \mathcal{Y}))) & \text{[Figure 5-3b]} \\
 & & & \\
\varrho(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \propto \mathcal{X}) \wedge (\varrho_1(\mathcal{X},\mathcal{Y}) \vee \varrho_2(\mathcal{X},\mathcal{Y})) & \\
\varrho_1(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{Y} \neq \mathrm{r}_\mathcal{X}) \wedge (\angle \mathrm{r}_\mathcal{Y}\mathrm{l}_\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{Y} \Downarrow \mathcal{X})) \vee & \text{[Figure 5-4a]} \\
 & & ((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{Y} \Uparrow \mathcal{X}))) & \text{[Figure 5-4b]} \\
\varrho_2(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{Y} = \mathrm{r}_\mathcal{X}) \wedge ((\angle\mathcal{X} < 180°) \vee (\angle\mathcal{Y} < 180°)) \wedge & \\
 & & (((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{Y} \Downarrow \mathcal{X})) \vee & \text{[Figure 5-5a]} \\
 & & ((\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{Y} \Uparrow \mathcal{X}))) & \text{[Figure 5-5b]} \\
 & & & \\
\tau(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\angle\mathcal{X} < 180°) \wedge & \\
 & & ((\mathrm{l}_\mathcal{X} \Uparrow \mathcal{Y}) \vee & \text{[Figure 5-6a]} \\
 & & (\mathrm{l}_\mathcal{X} \Downarrow \mathcal{Y})) & \text{[Figure 5-6b]} \\
 & & & \\
\psi(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \not\propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\angle\mathcal{X} < 180°) \wedge & \\
 & & ((\mathrm{r}_\mathcal{X} \Uparrow \mathcal{Y}) \vee & \text{[Figure 5-7a]} \\
 & & (\mathrm{r}_\mathcal{X} \Downarrow \mathcal{Y})) & \text{[Figure 5-7b]} \\
 & & & \\
\omega(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_\mathcal{X} \not\propto \mathcal{Y}) \wedge (\mathrm{r}_\mathcal{X} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_\mathcal{Y} \not\propto \mathcal{X}) \wedge (\mathrm{r}_\mathcal{Y} \not\propto \mathcal{X}) \wedge & \\
 & & ((\mathrm{l}_\mathcal{X} \neq \mathrm{l}_\mathcal{Y}) \vee & \text{[Figure 5-8]} \\
 & & ((\mathrm{l}_\mathcal{X} = \mathrm{l}_\mathcal{Y}) \wedge (\angle\mathcal{X} = 180°))) & \text{[Figure 5-9]}
\end{array}
$$

Table 5-2: Boolean expansion of $(\mathcal{X} \otimes \mathcal{Y})$

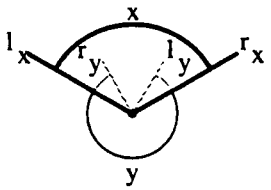


Figure 5-1a:
$((l_x \Uparrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge (l_y \Downarrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

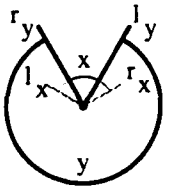


Figure 5-1b:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y}) \wedge (l_y \Uparrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

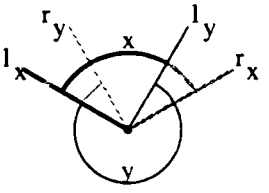


Figure 5-1c:
$((l_x \Uparrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y}) \wedge (l_y \Downarrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

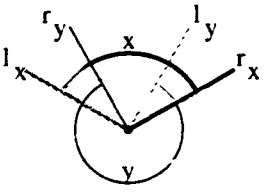


Figure 5-1d:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge (l_y \Uparrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

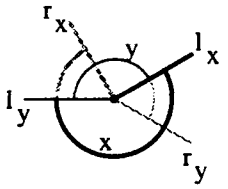


Figure 5-2a:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge (l_y \Uparrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

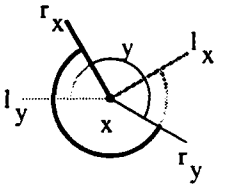


Figure 5-2b:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge (l_y \Downarrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

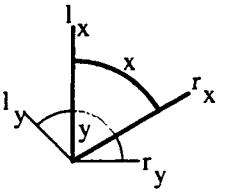


Figure 5-3a:
$(l_x \Uparrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y})$

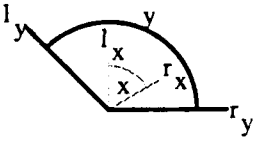


Figure 5-3b:
$(l_x \Downarrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y})$

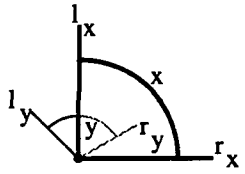


Figure 5-4a:
$(l_x \Uparrow \mathcal{Y}) \wedge (r_y \Downarrow \mathcal{X})$

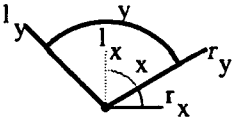


Figure 5-4b:
$(l_x \Uparrow \mathcal{Y}) \wedge (r_y \Downarrow \mathcal{X})$

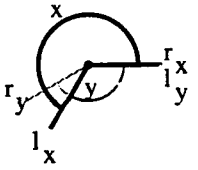


Figure 5-5a:
$(l_x \Uparrow \mathcal{Y}) \wedge (r_y \Downarrow \mathcal{X})$

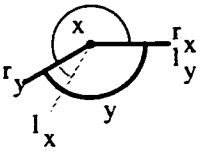


Figure 5-5b:
$(l_x \Downarrow \mathcal{Y}) \wedge (r_y \Uparrow \mathcal{X})$

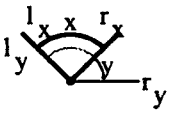


Figure 5-6a:
$(r_x \Uparrow \mathcal{Y})$

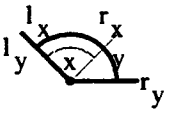


Figure 5-6b:
$(r_x \Downarrow \mathcal{Y})$

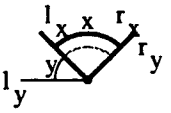


Figure 5-7a:
$(l_x \Uparrow \mathcal{Y})$

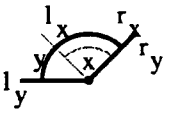


Figure 5-7b:
$(l_x \Downarrow \mathcal{Y})$

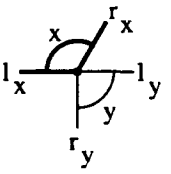


Figure 5-8:
(True)

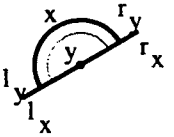


Figure 5-9:
(True)

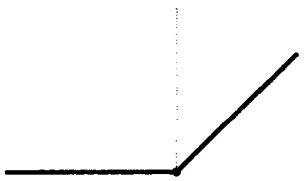


Figure 5-10: A vertex to be labeled.

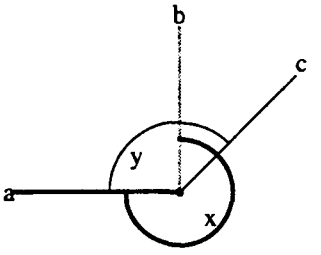


Figure 5-11: Proposed labeling for Figure 5-10.

$l_x = b$	$r_x = a$
$l_y = a$	$r_y = c$
$\angle a = 180°$	$\angle \mathcal{X} = 270°$
$\angle b = 90°$	$\angle \mathcal{Y} = 135°$
$\angle c = 45°$	
$V = \{a, c\}$	$H = \{b\}$
$(b \propto \mathcal{Y})$	$(c \propto \mathcal{X})$

Table 5-3: Data for Figure 5-11.

5.2.1 An example of testing a vertex's labeling

Suppose you have a vertex like the one shown in Figure 5-10 and wish to determine whether the labeling shown in Figure 5-11 is valid. The first step in determining the validity of the labeling, according to the procedure outlined in Section 5.2, is to translate the first order logic expressions from Section 5.2 into Boolean expression.

For Figure 5-11, the Boolean expressions corresponding to equations (5.1) – (5.3) are:

$$(c \Uparrow \mathcal{X}) \tag{5.1'}$$

$$((\{a,c\} = \{\emptyset\}) \vee (b \Downarrow \mathcal{Y})) \tag{5.2'}$$

$$((\mathcal{X} \otimes \mathcal{Y}) \wedge (\mathcal{Y} \otimes \mathcal{X})) \tag{5.3'}$$

Expression (5.2') can be rewritten as the following expression:

$$(b \Downarrow \mathcal{Y}) \tag{5.2''}$$

Since the $\otimes$ predicate is commutative, expression (5.3') is equivalent to:

$$(\mathcal{X} \otimes \mathcal{Y}) \tag{5.3''}$$

Using Table 5-2, this can be expanded to:

$$\begin{aligned}(\mathcal{X} \otimes \mathcal{Y}) \;=\; & \eta(\mathcal{X},\mathcal{Y}) \vee \kappa(\mathcal{X},\mathcal{Y}) \vee \varrho(\mathcal{X},\mathcal{Y}) \vee \tau(\mathcal{X},\mathcal{Y}) \vee \psi(\mathcal{X},\mathcal{Y}) \vee \omega(\mathcal{X},\mathcal{Y}) \vee \\ & \eta(\mathcal{Y},\mathcal{X}) \vee \kappa(\mathcal{Y},\mathcal{X}) \vee \varrho(\mathcal{Y},\mathcal{X}) \vee \tau(\mathcal{Y},\mathcal{X}) \vee \psi(\mathcal{Y},\mathcal{X}) \vee \omega(\mathcal{Y},\mathcal{X})\end{aligned} \tag{5.5}$$

Each term in expression (5.5) corresponds to a different pattern of spanned edges. The pattern of spanned edges in Figure 5-11 corresponds to $\varrho(\mathcal{X},\mathcal{Y})$ – all other terms are false. Therefore, in this particular case, $(\mathcal{X} \otimes \mathcal{Y}) = \varrho(\mathcal{X},\mathcal{Y})$.

Expanding $\varrho(\mathcal{X},\mathcal{Y})$ and evaluating everything except the $\Uparrow$ and $\Downarrow$ predicates produces:

$$\begin{aligned}\varrho(\mathcal{X},\mathcal{Y}) &= (b \propto \mathcal{Y}) \wedge (a \not\propto \mathcal{Y}) \wedge (a \not\propto \mathcal{X}) \wedge (c \propto \mathcal{X}) \wedge (\varrho_1(\mathcal{X},\mathcal{Y}) \vee \varrho_2(\mathcal{X},\mathcal{Y})) \\ &= (\varrho_1(\mathcal{X},\mathcal{Y}) \vee \varrho_2(\mathcal{X},\mathcal{Y}))\end{aligned}$$

$$\begin{aligned}\varrho_1(\mathcal{X},\mathcal{Y}) &= (a \neq a) \wedge (45° < 180°) \wedge (((b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee ((b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X}))) \\ &= (\text{False})\end{aligned}$$

$$\begin{aligned}\varrho_2(\mathcal{X},\mathcal{Y}) &= (a = a) \wedge ((270° < 180°) \vee (135° < 180°)) \wedge \\ &\quad (((b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee ((b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X}))) \\ &= (((b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee ((b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X})))\end{aligned}$$

$$\begin{aligned}\varrho(\mathcal{X},\mathcal{Y}) &= (\varrho_1(\mathcal{X},\mathcal{Y}) \vee \varrho_2(\mathcal{X},\mathcal{Y})) \\ &= ((\text{False}) \vee \varrho_2(\mathcal{X},\mathcal{Y})) \\ &= \varrho_2(\mathcal{X},\mathcal{Y}) \\ &= (((b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee ((b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X})))\end{aligned} \tag{5.3'''}$$

Combining (5.1′), (5.2″) and (5.3‴) produces:

$$(c \Uparrow \mathcal{X}) \wedge (b \Downarrow \mathcal{Y}) \wedge (((b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee ((b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X})))$$

Rewriting this expression in disjunctive normal form produces:

$$\begin{aligned}&((c \Uparrow \mathcal{X}) \wedge (b \Downarrow \mathcal{Y}) \wedge (b \Uparrow \mathcal{Y}) \wedge (c \Downarrow \mathcal{X})) \vee \\ &((c \Uparrow \mathcal{X}) \wedge (b \Downarrow \mathcal{Y}) \wedge (b \Downarrow \mathcal{Y}) \wedge (c \Uparrow \mathcal{X}))\end{aligned} \tag{5.6}$$

If this expression is satisfiable, then the labeling shown in Figure 5-11 is valid.

5.3 Determining the satisfiability of a validity expression

The satisfiability of an expression containing the $\Uparrow$ and $\Downarrow$ predicates can be evaluated if the expression can be written in the following form:

$$C = (a_1 \Uparrow \mathcal{X}_1) \wedge (a_2 \Uparrow \mathcal{X}_2) \wedge \cdots \wedge (a_n \Uparrow \mathcal{X}_n) \wedge (a_{n+1} \Downarrow \mathcal{X}_{n+1}) \wedge \cdots \wedge (a_m \Downarrow \mathcal{X}_m) \tag{5.7}$$

This expression is obviously unsatisfiable if it contains $(a_i \Uparrow \mathcal{X}_i)$ and $(a_j \Downarrow \mathcal{X}_j)$ terms in which $a_i = a_j$ and $\mathcal{X}_i = \mathcal{X}_j$. Potentially satisfiable expressions are evaluated using an algorithm similar to the one developed by Sugihara [27] to test the feasibility of a line drawing's interpretation.

The first step is to create vectors for each edge and then to find the normal vector for each surface fragment. These vectors and normals are created as follows:

- For each $a \in \bigcup_{i=1}^{m} \{a_i\}$, create a vector:
  $\vec{p}_a \Leftarrow (\cos \angle a, \sin \angle a, z_a)$
  where z_a is unknown ($\vec{p}_a$ is the position of the endpoint of edge a).

- For each $\mathcal{X} \in \bigcup_{j=1}^{m} \{\mathcal{X}_j\}$, create a vector:
  $$\vec{n}_{\mathcal{X}} \Leftarrow \begin{cases} (\vec{p}_{\mathrm{r}_{\mathcal{X}}} \times \vec{p}_{\mathrm{l}_{\mathcal{X}}}) & \text{if } (\angle\mathcal{X} < 180°) \\ (\vec{p}_{\mathrm{r}_{\mathcal{X}}} \times (\cos \angle \mathrm{r}_{\mathcal{X}} + 90°, \sin \angle \mathrm{r}_{\mathcal{X}} + 90°, \zeta_{\mathcal{X}})) & \text{if } (\angle\mathcal{X} = 180°) \\ (\vec{p}_{\mathrm{l}_{\mathcal{X}}} \times \vec{p}_{\mathrm{r}_{\mathcal{X}}}) & \text{otherwise} \end{cases}$$
  where $\zeta_{\mathcal{X}}$ is unknown ($\vec{n}_{\mathcal{X}}$ is an upward-pointing normal for the plane defined by $\mathcal{X}$).

The dot product of $\vec{p}_i$ and $\vec{n}_{\mathcal{X}}$ can be used to determine whether $\vec{p}_i$ is above, on, or below $\mathcal{X}$. If the dot product is positive, $\vec{p}_i$ is above $\mathcal{X}$; if it is zero, $\vec{p}_i$ is on $\mathcal{X}$; otherwise, $\vec{p}_i$ is below $\mathcal{X}$. Using this, the predicates in C can be evaluated as follows:

- For each predicate $(a_i \Uparrow \mathcal{X}_i)$, create the equation: $\vec{n}_{\mathcal{X}_i} \cdot \vec{p}_{a_i} > 0 \quad (1 \leq i \leq n)$
- For each predicate $(a_j \Downarrow \mathcal{X}_j)$, create the equation: $\vec{n}_{\mathcal{X}_j} \cdot \vec{p}_{a_j} < 0 \quad (n < j \leq m)$
- For each $\mathcal{X} \in \bigcup_{j=1}^{m} \mathcal{X}_j \mid \angle\mathcal{X} = 180°$, create the equation: $z_{r_{\mathcal{X}}} + z_{l_{\mathcal{X}}} = 0$

The last equation is needed because the normal vector for a surface fragment that spans exactly 180° was generated using the assumption that its bounding edges were parallel.

Even though the first two equations are linear, they are not in a form that can be solved using linear programming. These equations can be converted to forms that can be solved using linear programming by replacing > 0 and < 0 with $\geq \varepsilon$ and $\leq -\varepsilon$ respectively (ε is any positive constant). This modified system of equations is equivalent to the original system[1]. If this modified system of equations has a solution, then there is a way to arrange the edges and surface fragments so that the relations described in C are satisfied.

5.3.1 Continuing the example

The first term of expression (5.6) is clearly inconsistent (for example, c cannot be simultaneously in front of and behind $\mathcal{X}$). The second term can be simplified and written as:

$$(c \Uparrow \mathcal{X}) \wedge (b \Downarrow \mathcal{Y}) \tag{5.6'}$$

Using the algorithm given in Section 5.3, the following vectors and equations can be generated from Figure 5-11 and expression (5.6'):

$$\begin{aligned}
\vec{p}_a &\Leftarrow (\cos 180°, \sin 180°, z_a) & [\vec{p}_a &\Leftarrow (-1, 0, z_a)] \\
\vec{p}_b &\Leftarrow (\cos 90°, \sin 90°, z_b) & [\vec{p}_b &\Leftarrow (0, 1, z_b)] \\
\vec{p}_c &\Leftarrow (\cos 45°, \sin 45°, z_c) & [\vec{p}_c &\Leftarrow (\sqrt{2}/2, \sqrt{2}/2, z_c)]
\end{aligned}$$

$$\begin{aligned}
\vec{n}_{\mathcal{X}} &\Leftarrow \vec{p}_b \times \vec{p}_a & [\vec{n}_{\mathcal{X}} &\Leftarrow (z_a, -z_b, 1)] \\
\vec{n}_{\mathcal{Y}} &\Leftarrow \vec{p}_c \times \vec{p}_a & [\vec{n}_{\mathcal{Y}} &\Leftarrow (z_a sqrt2/2, -(z_a sqrt2/2 + z_c), sqrt2/2)]
\end{aligned}$$

$$\vec{n}_{\mathcal{X}} \cdot \vec{p}_c > 0 \qquad [z_a sqrt(2)/2 + z_b sqrt(2)/2 + z_c > 0] \tag{5.8}$$

$$\vec{n}_{\mathcal{Y}} \cdot \vec{p}_b < 0 \qquad [-(z_a sqrt(2)/2 + zc) + z_b sqrt(2)/2 < 0] \tag{5.9}$$

Equations (5.8) and (5.9) can be satisfied by making the following assignments:

$$\begin{aligned}
z_a &\Leftarrow 0 \\
z_b &\Leftarrow 0 \\
z_c &\Leftarrow 1
\end{aligned}$$

Since a solution that satisfies equations (5.8) and (5.9) exists and these equations were the generated from the validity criteria, the labeling shown in Figure 5-11 is valid. In other words, it is possible to build a physical model of the labeling shown in Figure 5-11 that matches the appearance of the vertex shown in Figure 5-10 and in which no two surfaces intersect except at a common boundary.

---

[1]Proof: A solution to the modified system of equations is a solution to the original system. Any solution to the original system can be multiplied a large enough constant that it constitutes a solution to the modified system. Therefore, if a solution exists to the original system, a solution exists to the modified system

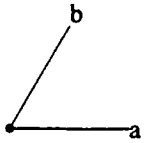


Figure 5-12a: Arrow/2. ($\angle ba > 180°$)

Figure 5-12b: Straight. ($\angle ba = 180°$)

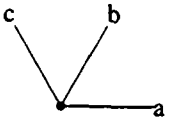


Figure 5-12c: Arrow/3. ($\angle ca > 180°$)

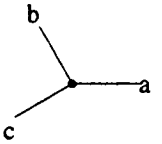


Figure 5-12d: Fork/3.

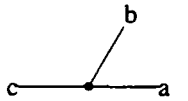


Figure 5-12e: T. ($\angle ca = 180° \wedge \angle ba > 180°$)

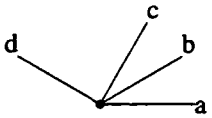


Figure 5-12f: Arrow/4. ($\angle da > 180°$)

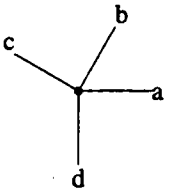


Figure 5-12g: Fork/4. ($\angle ca > 180° \wedge \angle bd > 180°$)

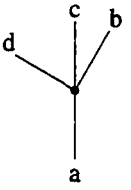


Figure 5-12h: Psi. ($\angle ca = 180° \wedge \angle db > 180°$)

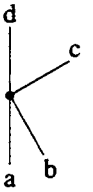


Figure 5-12i: K. ($\angle da = 180° \wedge \angle cb > 180°$)

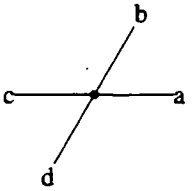


Figure 5-12j: X. ($\angle ca = 180° \wedge \angle bd = 180°$)

5.4 Vertex types

The valid labelings for a vertex are insensitive to the exact position of the edges around the vertex. In particular, two vertices will have the same *valid labelings if:*

- the corresponding pairs of edges around each vertex have the same angular sense (less than 180°, equal to 180° or greater than 180°) and
- the corresponding edges around each vertex are both either visible or hidden.

Although I do not have a formal proof for this conjecture, it seems plausible because:

- The disjunctive normal form of the validity criteria (see Section 5.2) for identical labelings on two similar vertices will be identical.
- Therefore, terms that are obviously inconsistent (see Section 5.3) for one vertex will be obviously inconsistent for the other.
- Also, the linear programming problems generated using the algorithm in Section 5.3 for each vertex will be similar. The only difference will be the angles used to generate the $\vec{p}_a$ and $\vec{n}_\chi$ vectors.
- Therefore, I believe that, since the LP consists primarily of inequality equations, the LP generated for one vertex will have a solution if and only if the corresponding LP for the other vertex has a solution.

If this assertion is true, then it is possible to create an intersection library by pre-computing the valid labelings for each vertex type. The advantage of using an intersection library is that it is faster find the valid labelings in the library than it is to generate them from scratch.

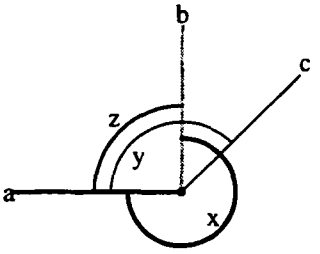


Figure 5-13a: A valid non-manifold labeling for the vertex in Figure 5-10.

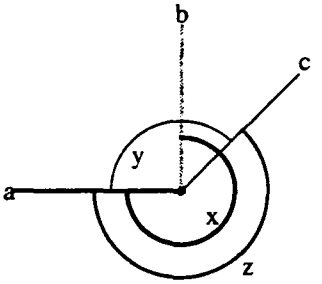


Figure 5-13b: An invalid labeling for the vertex in Figure 5-10.

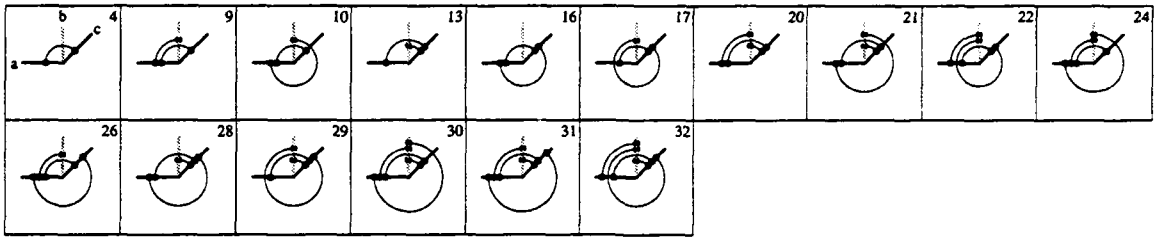


Figure 5-14: Valid labelings for Figure 5-10 in which edges can have 0 – 3 adjacent surfaces.

Figures 5-12a through 5-12j show all distinct arrangements of two, three or four edges around a vertex. The valid labelings for a vertex can be found by matching it against one of these arrangements and then finding the entry in the library that corresponds to the pattern of visible and hidden adjacent edges. Using the intersection library in Appendix D, for example, an Arrow/3 intersection in which every adjacent edge is visible has 19 valid labelings (compared to 23 valid labelings for an Arrow/3 intersection in which every adjacent edge is hidden).

5.5 Non-manifold surface topologies

An object with a manifold surface topology can be thought of as a "solid" object: it has a distinct inside and a distinct outside. It is, essentially, the three-dimensional equivalent of a two-dimensional closed polygon. And, although the majority of object design deals solely with manifold objects (solid modelers are called that for a reason), non-manifold objects have their uses.

In manifold objects, every edge is adjacent to an even number of faces. Huffman-Clowes line-labeling [7, 17] is limited to line drawings in which every edge in the corresponding object is adjacent to exactly two faces. Arc-labeling does not have this limitation: the validity criteria works even when a labeling calls for an edge to be adjacent to three or more surface fragments.

For example, consider the labeling in Figure 5-13a. This is identical to the labeling in Figure 5-11 except that a third surface, $\mathcal{Z}$ ($l_{\mathcal{Z}} = a$ and $r_{\mathcal{Z}} = b$), has been added. When testing the validity of the labeling in Figure 5-13a, equation (5.3') would include $(\mathcal{X} \otimes \mathcal{Z})$ and $(\mathcal{Y} \otimes \mathcal{Z})$ terms. The end result, however, would be the same: the labeling in Figure 5-13a is valid.

Suppose, as in Figure 5-13b, that $\mathcal{Z}$ had $l_{\mathcal{Z}} = c$ and $r_{\mathcal{Z}} = a$. This is because $\mathcal{X}$ and $\mathcal{Z}$ intersect

one another at someplace other than a bounding edge[2]. Since $\mathcal{X}$ and $\mathcal{Z}$ intersect, $(\mathcal{X} \otimes \mathcal{Z})$ is false and the validity criteria for the labeling can not be satisfied. Therefore is it not possible to create a model of the labeling whose physical appearance matches Figure 5-13b and the labeling is invalid.

Figure 5-14 shows all of the valid labelings for the vertex in Figure 5-10 if the restrictions on the number of faces adjacent to an edge were removed. Labeling 10 corresponds to the labeling shown in Figure 5-11 and labeling 24 corresponds to the labeling shown in Figure 5-13a. Since the labeling in Figure 5-13b is invalid, there is no corresponding labeling in Figure 5-14.

[2]Proof: Both surfaces contain edge a and, therefore, contain the line defined by edge a. Since both surfaces span over 180°, both surfaces must intersect along both a and its reciprocal vector.

Chapter 6

Viking's performance

Two critical aspects of an interactive system are how quickly it can react to the user's actions and how performance degrades as the problem size increases. *Viking's* performance as an interactive system depends on how quickly it can generate surface topologies and solve for vertex geometries, since these are the only operations that take a significant amount of processing time. Nearly 18 CPU seconds are required to find a surface topology for a line drawing with 100 vertices and over 70 CPU seconds might be needed to find a vertex geometry for an object with 100 vertices and 99 distance constraints. In both cases, performance is a non-linear function of the problem size.

6.1 Surface topology generation

Viking's performance at generating surface topologies was tested by creating several faceted spheres, such as the one shown in Figure 6-1, and modifying the line drawing by making all of the line-segments obscured by one face visible. Table 6-1 lists the times required to perform the steps required to find a surface topology corresponding to the modified line drawing on a Sun SPARC station 1+, a 16 MIPS workstation.

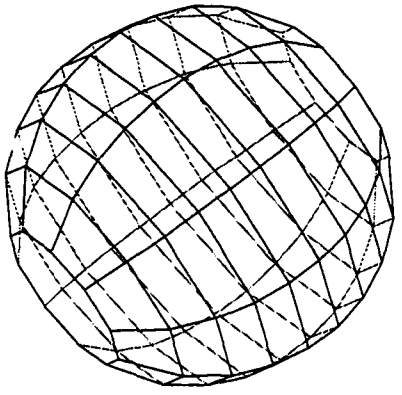


Figure 6-1: 100 vertex object for testing topology search performance.

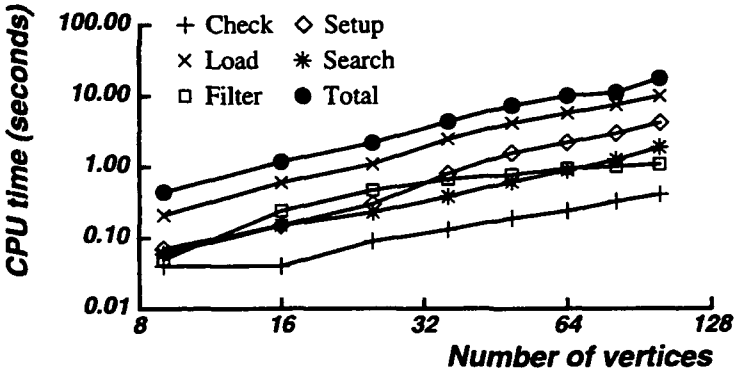


Figure 6-2: log/log graph of CPU time vs. number of vertices (topology generator).

# of vertices	9	16	25	36	49	64	81	100
# of labelings	121	375	707	1691	3013	3852	4777	6697
# of leaves	15	34	56	107	164	233	311	399
Total memory (MB)	0.7	1.3	2.0	3.8	6.3	8.4	11.0	14.6
Check time	0.04	0.04	0.09	0.13	0.19	0.24	0.33	0.41
Load time	0.21	0.60	1.10	2.46	4.19	5.75	7.58	9.98
Filter time	0.05	0.24	0.47	0.67	0.78	0.95	1.03	1.09
Setup time	0.07	0.15	0.30	0.80	1.58	2.22	3.00	4.21
Search time	0.06	0.15	0.23	0.38	0.62	0.90	1.27	1.89
Total time	0.44	1.18	2.19	4.36	7.36	10.06	11.23	17.58

# of labelings	is the total number of vertex labelings found in the load step.
# of leaves	is the total number of leaves in the search tree.
Total memory	is the amount of memory used for *Viking* during the search.
Check time	time required to check for obvious inconsistencies.
Load time	time required to load the valid labelings for each intersection.
Filter time	time required to perform Waltz-filtering [29].
Setup time	time required to initialize the search tree.
Search time	time required to search for the surface topology.
Total time	total time required to find the surface topology.

All times are in CPU seconds on a Sun SPARC station 1+

Table 6-1: CPU time requires to generate a surface topology.

Viking's memory requirements are high: over 14 megabytes to find an interpretation of the 100 vertex line drawing. As a result, *Viking* often spends a significant amount of time page-swapping, time that is not reflected in the CPU time. For example, finding an interpretation for the 100 vertex object takes, approximately, 20 "wall-clock" seconds on a SPARC station with 24 MB of RAM and almost 150 "wall-clock" seconds on one with only 16 MB of RAM.

The examples used in this performance analysis represent, to some extent, *Viking's* worst case performance. For example, *Viking* needs approximately the same amount of time to find an interpretation for the 175 vertex object in Figure 1-3d as it does to find an interpretation for the 100 vertex test object. Most of the test object's vertices are adjacent to four edges and have over 100 valid labelings apiece (see Appendix D). In addition, since there are few occluding edges or peripheral vertices, only a small fraction of the labelings are rejected by the crossing or perimeter rules (see Section 3.2). As a result, a large number of possible vertex labelings are found in the load step and, consequently, *Viking* spends most of its time loading the possible labelings for each vertex. This overhead can probably be reduced by changing the way labelings are "loaded." In particular, *Viking* creates a vertex specific instance of each appropriate labeling for each vertex. If, instead, *Viking* were to merely reference the labeling in the intersection library and calculate the vertex specific values on demand, the load step would be significantly faster.

In its current form, *Viking* requires less than 18 CPU seconds to find an interpretation for a line drawing like the one shown in Figure 6-1. Also, the CPU time seems roughly proportional to $n^{1.5}$ (where n is the number of vertices), at least for the problem sizes used. The algorithm used to find a surface topology is one that should exhibit exponential growth. But, at least for problems with

# of vertices		9	16	25	36	49	64	81	100
Distance	Min time	0.13	0.26	0.61	1.89	2.34	5.51	11.16	27.01
	Max	0.16	0.46	0.98	2.61	8.96	21.23	29.74	73.34
Planar	Min time	0.23	0.34	0.56	0.93	1.49	2.83	3.48	5.28
	Max	0.29	0.54	0.74	1.26	2.26	3.21	4.88	7.89
Angular	Min time	0.33	0.74	1.61	2.49	4.24	7.38	10.99	16.13
	Max	0.41	0.98	2.38	3.61	6.69	9.61	22.98	31.29

Min time best case time required to find a satisfactory geometry.
Max time worst case time required to find a satisfactory geometry.
Times are in CPU seconds on a Sun SPARC station 1+
Table 6-2: CPU time requires to solve a system of constraints.

Figure 6-3a: 9 vertex distance constraint network.

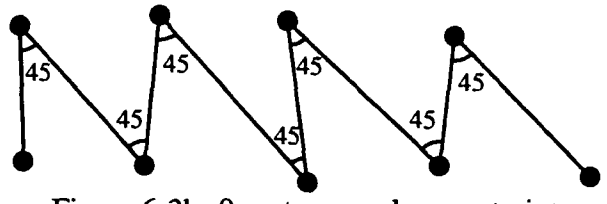


Figure 6-3b: 9 vertex angular constraint network.

less than 100 vertices, the exponential components of the search are dominated by the polynomial ones. With more efficient algorithms and faster workstations, arc-labeling should allow future versions of *Viking* to provide response times appropriate for an interactive system, especially when used with relatively simple objects.

6.2 Constraint satisfaction

Another aspect of *Viking's* performance is the amount of time required to find a geometry that satisfies a set of geometric constraints. Unfortunately, since different systems of constraints have radically different behavior, *Viking's* performance in this area is harder to characterize than its performance when generating surface topologies. However, by looking at *Viking's* performance on several different constraint networks, it is possible to draw some general conclusions.

The following three distinct constraint networks, each using a different type of geometric constraint, were used to analyze the performance of *Viking's* constraint solver (where i is the vertex ID and $[-0.1, 0.1]$ is a randomly generated number between -0.1 and 0.1):

- Distance: every vertex is constrained to be exactly one unit away from its left and right neighbors (see Figure 6-3a). The initial vertex positions are:
$$(i + [-0.1, 0.1], [-0.1, 0.1], [-0.1, 0.1])$$
- Planar: every vertex is constrained to be co-planar with every other vertex. The initial vertex positions are:
$$(10[-0.1, 0.1], 10[-0.1, 0.1], [-0.1, 0.1])$$

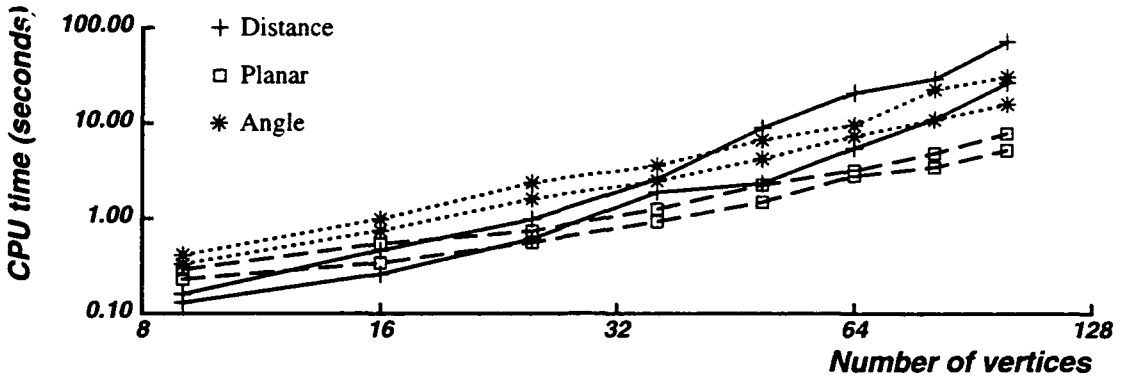


Figure 6-4: log/log graph of CPU time vs. number of vertices (constraint solver).

- Angular: the angle formed by every vertex's left neighbor, itself and its right neighbor is constrained to be 45° (see Figure 6-3b). The initial vertex positions are:
$$([i/2] + [-0.1, 0.1], (i \bmod 2) + [-0.1, 0.1], [-0.1, 0.1])$$

Each combination of network type and problem size was run four times, using different initial vertex positions for each run. The results are shown in Table 6-2 and Figure 6-4.

Solving for a vertex geometry is, at least with these networks, even slower than finding a surface topology: over 70 CPU seconds were required in one instance. CPU time seems to be roughly proportional to $n^{2.5}$ (where n is the number of vertices). Fortunately, it is possible to improve the constraint solver's performance by locking vertices and using components to reduce the network's complexity. This is not, however, a perfect solution since even simple systems can take unacceptably long to solve and it is not always clear which vertices to lock or how to use components effectively.

Viking's constraint solver is used in two different ways. The first is when the constraint network has changed, perhaps because the user added a new constraint. The second is when the user is interactively dragging a vertex to a new position while maintaining the existing constraints. These two cases are fundamentally different from one another. In the latter, the constraints are normally satisfied before the user starts dragging the vertex and performance is critical to providing an illusion of smooth motion. In the former, some constraints are initially violated and performance is not as important as it is when the user is dragging a vertex.

The performance of *Viking's* constraint solver seems adequate for the cases when the constraint network has changed. It is, however, too slow when dragging a vertex in anything but simple constraint networks. It may be possible to improve "dragging" performance by using a different constraint solver when the user is dragging a vertex. In particular, differential constraints [12] seem to work well when the constraints are initially satisfied. In addition, differential constraints have the advantage that the "dragging constraints" (see Section 4.1.3) can be explicitly represented as a force on the dragged vertex that "pulls" the vertex closer to the position indicated by the mouse.

Chapter 7

Conclusions

Viking is a solid modeling system that uses interactive sketch interpretation to combine the simplicity of pencil and paper sketches with the power of a solid modeling system. *Viking* lets designers draw the object they wish to create and then modify it by changing the line drawing to make it "look right." Each action is obvious from context, leaving the designer free to concentrate on the design itself and not how to convey it to the solid modeler.

This ease of use comes without sacrificing any of the capabilities intrinsic to solid modeling systems. As with other solid modeling systems, *Viking* lets the designer manipulate the underlying object description as if it were a solid object. This provides the designer with a powerful tool for visualizing an object's structure. For example, the designer can wiggle the object by dynamically changing the view transform or drag a translucent cutting plane through the object to see where vertices lie with respect to one another in three dimensions. And, although *Viking's* user interface is based primarily on sketching, the designer can create precisely dimensioned models by using geometric constraints. This combination of sketching and solid modeling techniques creates an effective design methodology for developing ideas into practical designs.

7.1 Accomplishments

This thesis sets out to demonstrate and explain a new type of user interface for solid modeling systems. This user interface uses interactive sketch interpretation and lets designers use the techniques normally used to make paper and pencil sketches to design solid objects. Interactive sketch interpretation makes it easier to use solid modelers to explore new ideas and refine these ideas into workable designs without sacrificing any of their intrinsic capabilities.

This user interface lets a designer sketch on the computer in much the same way that he or she would sketch on pencil and paper. Sketch interpretation is the process of using the designer's actions to modify a description of a three-dimensional object so that the sketch and the object are mutually consistent. Sketch interpretation in *Viking* takes a wide variety of forms. At one extreme, it is simply a matter of giving the designer the tools to position vertices in three space. At the other extreme, it is a matter of finding a new object description that is consistent with a line drawing and a set of geometric constraints.

In the process of developing *Viking*, I have developed a set of algorithms and user interface techniques that advance the state of the art. Arc-labeling, used to do topology generation, extends

the line-labeling methodology [7, 17] to non-trihedral objects. And, although *Viking* is limited to objects in which every vertex is adjacent to four or fewer edges and every face is a planar polygon, these limits are not intrinsic to arc-labeling. *Viking* is one of a small number of 3D design systems that do constraint satisfaction. *Viking's* algorithm for constraint satisfaction is relatively fast, robust and capable of supporting a wide variety of constraints. *Viking* is also the only 3D design system based on the premise that the designer is sketching: making a "quick and dirty" drawing and then gradually refining it by making small, incremental changes.

7.2 Future work

Viking's user interface has some significant weaknesses. Some of these are problems that should not be difficult to solve. Others do not seem to have easy solutions. These problems are presented in the order that they will be addressed in future research.

CAD modeling interface

Currently, *Viking* provides few of the capabilities found in conventional solid modeling systems. For example, *Viking* can neither calculate the mass of an object nor find the intersection of two objects. Combining conventional solid modeling capabilities and interactive sketch interpretation should not be difficult: *Viking's* underlying object description is equivalent to the boundary representation description used by some solid modelers.

Explicit constraints specification

Viking's users must explicitly specify every geometric constraint. Other constraint based design systems, such as Gargoyle [1] and Briar [11], provide mechanisms for defining constraints implicitly. Incorporating similar mechanisms into *Viking* could alleviate one of the more tedious aspects of *Viking's* user interface.

Planar faces and straight edges

Viking can, currently, only interpret line drawings of objects whose faces are all planar polygons. The topology generation algorithm can be extended to objects with non-planar faces. Modifying the rest of *Viking*, however, is more difficult. Planar faces provide one of the better implicit constraints, and designing a good user interface for specifying which faces are non-planar and controlling the shape of a non-planar face is not easy.

It should, however, be comparatively easy to extend *Viking* to allow certain types of non-planar faces. In particular, if the user were allowed to create curved edges, then *Viking* could generate surface topologies in which a face is considered non-planar if it is bounded by one or more curved edges. For example, the user could define a circular "arc edge" using three vertices. These edges could then be used to define the faces forming a cylinder.

Quadhedral vertices

Viking can only analyze line drawings in which every vertex is adjacent to four or fewer edges. This is because *Viking's* topology generation algorithm must match every intersection in the line drawing to an entry in a fixed intersection library that contains all possible labelings, or configurations of faces, around the intersection. The program used to generate this library, however, is already able to generate both entries for intersections of

five or more lines and entries in which edges can be adjacent to three or more faces. *Viking* can be modified to use this algorithm to compute the labelings for any intersection not in the intersection library.

Simple polygonal faces

Faces in *Viking* must be simple, planar polygons: they cannot have internal holes, repeated edges or repeated vertices. It should be possible to extend the algorithm to allow more complicated faces, although it may not be worth the extra processing time required. The current version of *Viking* lets the user simulate holes and the like by using artifact edges.

Explicit topology specification

Viking's topology generation algorithm uses the presence of hidden line-segments to automatically reject inconsistent interpretations. The downside of this is that the user must correctly indicate which line-segments are hidden. This can be a tedious and time-consuming process.

Viking lets the user generate a blind interpretation, in which the visibility cues are ignored and the user does not have to indicate which line-segments are hidden. Blind interpretations are slower and less discriminating than conventional interpretation, since visibility cues cannot be used to reject unwanted topologies. Despite this, it is often easier to generate a blind interpretation and manually reject unwanted topologies than it is to indicate which line-segments are hidden and generate a standard interpretation.

7.3 Open problems

The following section describes problems that do not seem to have easy solutions.

General view

Viking's topology generation algorithm can only interpret line drawings that correspond to a general view of an object. A general view is one in which a small change in the view direction makes correspondingly small change in the line drawing [24]. For example, a general view could not contain any faces that are "edge-on" to the viewer (such as Figure 2-3j).

This is a problem since engineering drawings do not always correspond to general views. However, it is not clear how significant this problem is. Engineering drawings often used specialized viewpoints because they were easier to draw or because they illustrated a particular point. Specialized views are not, for the most part, easier to interpret than general views and both types of views are easy to generate using the computer.

One possibility for generating interpretations of specialized views is to use graph based algorithms [8, 16]. These algorithms do not depend on the viewpoint, generating a surface topology by finding a planar embedding of an object's vertex-edge graph. Unfortunately, it may not be possible to modify these algorithms to use heuristics to generate non-manifold topologies.

Topology generation performance

Viking's topology generation algorithm is not as fast as one might wish, taking almost three minutes to generate an interpretation of a line drawing containing 100 points. However most of the time seems to be spent page swapping (see Section 6.1). Although faster workstations and more efficient algorithms may alleviate this problem, it is not realistic to expect that *Viking's* topology generation algorithm could be used on large objects (which might be three or four orders of magnitude more complex than the objects created in Sections 2.2.1 or 2.2.2). It should, however, be possible to automatically partition a large object and use topology generation on only the relevant parts.

Constraint satisfaction performance

Viking's constraint solver is used in two basic modes: when one or more constraints have been added and *Viking* must solve for a solution and when the user is moving a vertex by dragging it with the mouse and wishes to maintain the pre-existing constraints. The response time when dragging is far slower than desired, often taking several seconds to find a solution that satisfies all the constraints. It might be possible to use differential constraints [12] to improve response times when dragging.

Constraint satisfaction robustness

Viking's constraint solver finds a satisfactory vertex geometry most of the time. There are some constraint networks, however, for which *Viking's* constraint solver cannot find a solution, even though there is a vertex geometry that satisfies all the constraints. In these situations, the user is often reduced to either moving vertices in the hopes that *Viking* will be able to find a solution from a different initial configuration, or rewriting the constraints in the hopes that *Viking* will be able to find a solution to a different but equivalent constraint network.

Appendix A

Glossary

appropriate labeling A labeling that is both valid for an intersection and consistent with the restrictions generated from features in the line drawing.

component A group of vertices that form a distinct group within the object. The object space coordinates of each of a component's vertices is a function of their position in the component space and the component's coordinate transform.

component space A coordinate system defined by a component's coordinate transform. The position of a vertex that is a member of a component is defined in terms of the component's coordinate system.

consistent labeling A labeling for the entire line drawing in which a labeling has been selected for each vertex and all of the selected labelings are consistent with one-another. Two labelings are consistent if there is no edge between the labeling's vertices or if the edge between the vertices is consistently labeled. An edge is consistently labeled if its two labels agree on the number of surfaces extending to either side of the edge.

crossing A place in the line drawing where two edges cross each other in the drawing without intersecting in three dimensions. For example: edges V_aV_c and V_bV_d cross each other in Figure 3-1 without intersecting.

cutting plane An arbitrary plane in object space. Used to help position the cursor in three dimensions.

edge An edge between two vertices. Edges can have zero, one or two adjacent faces.

explicit constraint A geometric constraint placed by the user. These constraints are always part of the system of constraints used to solve for a vertex geometry.

face A planar polygon bounded by three or more vertices. Faces can be bounding faces (i.e. defining a boundary between the "inside" of the object and the "outside") or be "thin-shell" faces (i.e. both sides of the face are on the "outside" of the object).

free vertex A vertex whose coordinates can be manipulated by the constraint solver when solving for a vertex geometry.

image constraint A geometric constraint derived from features in the line drawing. These constraints are only part of the system of constraints used to solve for a vertex geometry immediately after searching for a new surface topology.

image space The coordinate space corresponding to the image displayed to the user. In this coordinate space, the X-axis is horizontal, the Y-axis is vertical and the Z-axis is into or out of the screen.

implicit constraint Either an image or a world constraint.

intersection A place in the line drawing where two or more edges meet at a vertex.

label A description of the surface fragments immediately adjacent to an edge near one of its vertices. In *Viking* an edge's label is a list of surface fragments extending to either the left or right of an edge and, for each surface fragment, a pointer to the surface fragment's other bounding edge.

labeling A list of labels for all of the edges adjacent to a vertex.

legitimate surface topology A surface topology in which no face contains a repeated edge or vertex.

line Equivalent to an edge.

line-segment A portion of an edge between two crossings or intersections. The user can set the visibility of each line-segment in the line drawing independently.

locked vertex A vertex whose coordinates are treated as fixed constants by the constraint solver.

object space The coordinate space normally used to define the position of the object's vertices.

preferred direction One of a set of vectors used to help the user position the endpoint of new edges in three dimensions. By default, new edges are projected onto the closest preferred direction.

pseudo-variable A variable whose value is a function of one or more variables. For example, a vertex's image space coordinates are pseudo-variables that are functions of the vertex's object space coordinates and the view transform.

reasonable surface topology A surface topology that is legitimate, consistent with the features in the line drawing and does not contain any spanning edges.

screen space Equivalent to image space.

spanning edge An edge that extends across the inside of a face between two of the face's vertices. The presence of spanning edges disqualify a surface topology since the constraint solver will not be able to satisfy the image constraints generated for the edge with respect to the face.

surface Equivalent to a face.

surface fragment A portion of a face, typically defined by two bounding edges and a common vertex. Surface fragments are joined together to form larger surface fragments and, eventually, closed polygons when generating an object's surface topology from a consistent labeling.

surface topology A description of all of the faces forming the object. Each face description consists of a list of the vertices that form the face's bounding polygon.

tack An explicit constraint that either forces a vertex to have a fixed location in object space (even if the vertex is a member of a component), or forces an edge to pass through a fixed location in space.

true variable An independent variable that can be manipulated by the constraint solver.

valid labeling A labeling for an intersection that is listed in *Viking's* intersection library (see Appendix D).

vertex A position in object space. Vertices can have $0-4$ adjacent edges.

vertex geometry The positions of all of the object's vertices.

world constraint A geometric constraint generated from the limitations on *Viking's* object representation. In particular, all faces are constrained to be planar polygons. These constraints are always part of the system of constraints used to solve for a vertex geometry.

Appendix B

Viking's user interface primitives

Viking lets the user manipulate the object and program state in a wide variety of ways. This section lists all of these actions and their effect on the object and program state.

B.1 View actions

The view control window is divided into a panel of thirteen buttons, five "dial" or slider controls and virtual trackball. The functions of each of these buttons is described below:

Solve Invoke the sketch interpreter or constraint solver.

The effect of picking Solve depends on which mouse button was used.

Left button Invoke the sketch interpreter.

Search for a new surface topology that is consistent with the line drawing displayed in the image window. If a surface topology is found and accepted by the user, use the constraint solver to find a vertex geometry that satisfies the implicit and explicit constraints.

Center button Invoke "blind" sketch interpretation.

Blind sketch interpretation is identical to normal sketch interpretation except that all visibility cues are ignored. The effect is identical to hiding the entire object behind some obscuring surface.

Right button Invoke the constraint solver.

Use the constraint solver to find a vertex geometry that satisfies the world and explicit constraints. Note that the image constraints are not used when this button is used

Split/Kill Create a new, independent view of the object/ delete the selected view.

If only one view is displayed, copy it and display both views until the user kills one of them. Although the view are initially identical, the user can manipulate each view independently.

If two views are displayed, kill the selected view. There is no fundamental reason that *Viking* cannot display more than two different views of the object.

HLs Toggle the display of hidden line-segments and vertices.

Faces Toggle the display of faces. The faces, if on, are drawn as filled gray polygons before any edges or vertices are drawn.

IDs Toggle the display of vertex IDs.

Cons Toggle the display of constraints.

PD Enter the preferred direction (see Section 2.1.1) sub-menu. Within this menu the user can:

- Select two vertices and *Viking* will add the vector between to the list of preferred directions.
- Add the object space X, Y and Z axes to the list of preferred directions.
- Put preferred directions on automatic so that context dependent preferred directions will be created when the user starts to draw a new edge.
- Turn preferred directions off.

Top Change the view transform to a view straight down the Y-axis. If this button was picked using the right mouse button, rotate the current view transform 90° about the screen's horizontal axis.

Front Change the view transform to a view straight down the Z-axis.

Right Change the view transform to a view straight down the X-axis. If this button was picked using the right mouse button, rotate the current view transform 90° about the screen's vertical axis.

CP Enter the cutting plane (see Section 2.1.1) sub-menu. Within this menu the user can:

- Activate the cutting plane. The cutting plane defaults the screen's plane, but can also be positioned so that is the plane defined by any three vertices.
- Turn the cutting plane off.
- Toggle the display of the cutting plane's grid.
- Toggle the display of shadows (an orthogonal projection of the object's edges onto the cutting plane).
- Toggle the display of poles (lines between all vertices and their orthogonal projection on the cutting plane).
- Toggle the display of cuts (the intersection between faces and the cutting plane).
- Toggle whether the cutting plane is opaque.
- Change the view transform so that the cutting plane lies in the plane of the screen.

⊕ Change the translation and scaling components of view transform so that the entire object is displayed in the image window.

- Enter the pan and zoom sub-menu. Within this menu, the user can pick two corners of a rectangle in the image window and the translation and scaling components of view transform so that the entire object is displayed in the selected rectangle.

  If this button was picked using the center mouse button, let the user change the translation component of the view transform by dragging the image up/down and left/right.

The virtual trackball, also called a continuous XY controller [5], lets the user change the rotation component of the view transform by dragging the mouse across the triad, the left-right dial or the up-down dial while holding one of the buttons down. If the mouse started on the triad, the image will rotate about an axis in the screen plane that is perpendicular to the mouse's motion. If the mouse started on the left-right dial, the image will rotate about the screen's vertical axis. And, if the mouse started on the up-down dial, the image will rotate about the screen's horizontal axis.

The image controller window also contains two sliders. The first slider lets the user adjust the view transform's scaling component, the amount of face shrink and the view transform's perspective distance using, respectively, the left, center and right buttons. Face shrink changes the way that *Viking* draws the objects faces. If face shrink is on, then the object's faces are drawn as closed polygons that are offset in from the face's vertices by an amount proportional to the face shrink. The second slider lets the user drag a cutting plane through the object without changing the cutting plane's orientation.

B.2 Image actions

The effect of the user's actions in the image window depends on the currently selected command mode. The four most commonly used command modes, *Edit, Move, Constrain* and *Component*, use the following mapping between action and effect:

Edit **mode** This mode is used to edit the structure of the object being designed. Using the mouse in the Image window has the following effect, depending on which mouse button is pressed:

Left button Start drawing new edge. Stop when the user releases all mouse buttons.

If the user starts and ends on the same vertex, toggle that vertex's locked status.

Center button The selected edge or vertex for deletion. If that edge or vertex is already marked, delete it.

Right button Toggle the visibility of the selected vertex or line-segment.

Move **mode** This mode is used to move vertices and edges to new positions. Using the mouse in the Image window has the following effect, depending on which mouse button is pressed:

Left button Drag the selected edge or vertex as long as a button on the mouse is held down. If the Autosolve switch is set to automatic solve, attempt to maintain all of the geometric constraints.

Center button If the user selected a vertex, toggle the locked status of the vertex.

If the user selected a vertex that is a member of a component, place a tack on the vertex to prevent it from moving.

If the user selected an edge, place a tack on the edge at the selected location. The tack constraints the edge to pass through the tack's location.

Right button The same as for the left button, but do not attempt to maintain the constraints even if the Autosolve switch is set to automatic solve.

Constraint **mode** This mode is used to place geometric constraints on the object. Using the mouse in the Image window has the following effect, depending on which mouse button is pressed:

Left button Add the selected vertex to the current constraint template. If the user selected an edge, add both of its vertices instead.

Viking provides the following constraint templates (d, k and a are constants that can be set by the user):

- $\mid v_a, v_b \mid = d$
  Constrain the distance between v_a and v_b to be d.
- $\mid v_a, (v_b, v_c) \mid = d$
  Constrain the distance between v_a and the line defined by $v_b v_c$ to be d.
- $\mid v_a, v_b \mid = k \mid v_c, v_d \mid$
  Constrain the distance between v_a and v_b to equal k times the distance between the v_c and v_d.
- $\mid v_a, (v_b, v_c) \mid = k \mid v_d, v_e \mid$
  Constrain the distance between v_a and the line defined by $v_b v_c$ to equal k times the distance between v_d and v_e.
- $\mid v_a, (v_b, v_c) \mid = k \mid v_d, (v_e, v_f) \mid$
  Constrain the distance between v_a, and the line defined by $v_b v_c$ to equal k times the distance between v_d and the line defined by $v_e v_f$.
- Angle$(v_a, v_b, v_c) = a$
  Constrain the angle formed by v_a, v_b, and v_c to be a degrees.
- Angle$(v_a, v_b, v_c, v_d) = a$
  Constrain the angle between the line defined by $v_a v_b$, and the line defined by $v_c v_d$ to be a degrees.
- Planar$(v_a, v_b, \ldots)$
  Constrain v_a, v_b, ... to lie in the same plane.
- Equilateral(v_a, v_b, v_c)
  Constrain v_a, v_b, and v_c to form an equilateral triangle.
- Rectangular(v_a, v_b, v_c, v_d)
  Constrain v_a, v_b, v_c, and v_d to form a rectangle.

Center button Reset the current constraint template.

Right button Add the constraint under construction to the object's list of explicit constraints.

Component **mode** This mode is used to manipulate components. Using the mouse in the Image window has the following effect, depending on which mouse button is pressed:

Left button Add the selected vertex to the current component. If an edge was selected, add both of its vertices.

Center button Remove the selected vertex from its component. If an edge was selected, remove both of its vertices from their component.

Right button Make the component containing the selected vertex the current component.

B.3 Menu actions

Viking continuously displays two menus. The first menu, displayed in the center of the screen, changes to reflect the current command mode. The second menu, displayed at the bottom center of the screen, is fixed. The following options are available from the primary variable menu (as shown in Figure 2-1):

Edit Enter the edit command mode.

Move Enter the move command mode.

Constraint Enter the constraint command mode and replace the variable menu with the constraint menu.

Component Enter the component command mode and replace the variable menu with the component menu.

The following options are available from the fixed menu:

Help off/on Toggle the display of the help window.

Undo Move all the vertices to the positions they had before the constraint solver was last used.

Write Write the current object description to a file.

Read Read a new object description from a file.

Autosolve Let the user change the Autosolve switch. This switch can have four values:

No solve/no search Do not attempt to maintain the geometric constraints or automatically search for an interpretation after a change.

No solve/automatic search Do not attempt to maintain the geometric constraints, but automatically search for a new interpretation whenever the user hides or exposes a line-segment.

Automatic solve/no search Attempt to maintain the constraints as new constraints are added or the user drags vertices around, but do not automatically search for an interpretation after a change.

Automatic search and solve Automatically maintain the constraints and search for a new interpretation.

Bias Let the user set the search bias. This switch affects the cost assigned to labelings (see Section 3.3.1) and can have three settings:

No bias Do not favor any particular type of surface topology.

Surface bias Favor surface topologies in which every has one or two adjacent surfaces.

Solid bias Favor surface topologies in which every edge has two adjacent surfaces.

Quit Exit *Viking*.

B.4 Help actions

Viking's help window (the right hand window in Figure 2-1) provides an explanation of the various buttons and command modes. The help text is divided into two parts: a variable part that depends on the currently selected command mode and a fixed part. Each part is arranged into a tree-like data structure, where each line of text corresponds to a node in the tree. Clicking on a line of text expands the corresponding node, causing the text corresponding to its children to be displayed below and to the right of the selected line. Clicking again on a line collapses the corresponding node, suppressing the display of its children.

Appendix C

Equations

This appendix describes all of the equations used to represent both geometric constraints and the transformations between object space and image space, and component space and object space. The geometric constraints are generally applied to the object space coordinates of a vertex. There are two exceptions, however: image constraints and component transforms.

Image constraints are applied to the image space coordinates of a vertex. The image space coordinates are simply functions of the object space coordinates and the view transform, which is fixed. It is, therefore, possible to rewrite any constraint on a vertex's image space coordinates as a constraint on its object space coordinates.

If a vertex is a member of a component, then its object space coordinates are a function of its component space coordinates (which are fixed) and the component transformation variables (which are not). As with the image constraints, constraints on the vertex's object space coordinates can be rewritten as a constraint on the component's transformation variables. In essence, this means that *Viking* satisfies the geometric constraints on a member of a component by moving or scaling the entire component.

C.1 Geometric constraints

Currently, *Viking* supports five different types of constraints. These constraints are listed below, along with the equations used to represent them. Each equation has two forms: an evaluation form used to generate the linear programming problem (see Section 4.4.1) and an error form used to find the "optimal" displacement (see Section 4.4.2).

The evaluation equations are designed to have simple derivatives. This makes calculating the derivatives faster and tends to reduce problems due to singularities. Error functions are designed to calculate an "error" that is roughly proportional to the distance the vertices must move in order to satisfy the constraint.

- Planarity constraint:

  Planar($\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$):
  $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ all lie in the same plane.

  To add a planarity constraint, create four new variables: a, b, c, d,
  add the equation:
  $a^2 + b^2 + c^2 = 1$, and
  for each i such that $1 \leq i \leq n$, add either the equation:
  $(\vec{v}_i \cdot (a, b, c)) + d = 0$ or the equation
  $(\vec{\vartheta}_i \cdot (a, b, c)) + d = 0$.
  Where:
  $\vec{\vartheta}_i$ is the image space coordinates of $\vec{v}_i$.
  The second form of the equation is used when a, b, c and d are needed for a Point *in front of* plane constraint.

  The corresponding error functions are:
  Error $= \mid a^2 + b^2 + c^2 - 1 \mid$
  and:
  Error = $\mid (\vec{v}_i \cdot (a, b, c)) + d \mid$ or
  Error = $\mid (\vec{\vartheta}_i \cdot (a, b, c)) + d \mid$.

- Point *in front of* plane constraint:

  PointFront($\vec{v}, a, b, c, d$) :
  $\vec{v}$ is in front of the plane defined by a, b, c, d
  (a, b, c, d are the variables from the planarity constraint, above).

  To add a point *in front of* plane constraint, add the equation:
  $(\vec{\vartheta} \cdot (a, b, c) + d)/c \geq \varepsilon$
  Where:
  $\vec{\vartheta}$ is the image space coordinates of $\vec{v}$, and
  ε is the minimum separation distance.

  The corresponding error function is:
  Error = $\max(0.0, \varepsilon - (\vec{\vartheta} \cdot (a, b, c) + d)/c)$

- Line *in front of* line constraint:

  LineFront($\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4$) :
  Line $\vec{v}_1\vec{v}_2$ passes in front of line $\vec{v}_3\vec{v}_4$.

  To add a line *in front of* line constraint, add the equation:
  $(C \cdot (\vec{\vartheta}_1 - \vec{\vartheta}_3))/C_z \geq \varepsilon$

Where:

$C = ((\vec{\vartheta}_2 - \vec{\vartheta}_1) \times (\vec{\vartheta}_4 - \vec{\vartheta}_3))$,
C_z is the z-component of C,
$\vec{\vartheta}_i$ is the image space coordinates of $\vec{v}_i$, and
ε is the minimum separation distance.

The corresponding error function is:

$$\text{Error} = \max(0.0, \varepsilon - (C \cdot (\vec{\vartheta}_1 - \vec{\vartheta}_3))/C_z)$$

- Distance constraint:

$$D(k_1, \vec{v}_1, \vec{v}_2, \ldots, k_n, \vec{v}_{2n-1}, \vec{v}_{2n}, k_{n+1}, \vec{v}_{2n+1}, \vec{v}_{2n+2}, \vec{v}_{2n+3}, \ldots, k_{n+m}, \vec{v}_{2n+3m-2}, \vec{v}_{2n+3m-1}, \vec{v}_{2n+3m}) \begin{cases} \leq & d \\ = & d \\ \geq & d \end{cases}$$

The distances between the given pairs of vertices must satisfy the linear equation shown below.

To add a distance constraint, add the following equation:

$$k_1 \mid \vec{v}_1 - \vec{v}_2 \mid + \ldots + k_n \mid \vec{v}_{2n-1} - \vec{v}_{2n} \mid + k_{n+1} L(\vec{v}_{2n+1}, \vec{v}_{2n+2}, \vec{v}_{2n+3}) + \ldots + k_{n+m} L(\vec{v}_{2n+3m-2}, \vec{v}_{2n+3m-1}, \vec{v}_{2n+3m}) \begin{cases} \leq & d \\ = & d \\ \geq & d \end{cases}$$

Where:

$L(v_a, v_b, v_c) = \mid (v_a - v_b) \times (v_c - v_b) \mid / \mid v_c - v_b \mid$
($L(v_a, v_b, v_c)$ is the distance between v_a and the line defined by v_b and v_c).

The corresponding error function is either the amount by which the distance constraint is not satisfied or 0 if the constraint is satisfied.

- Angular constraint:

$\text{Angle}(\vec{v}_p, \vec{v}_1, \vec{v}_2, \vec{v}_3) = a$

The angle between lines $\vec{v}_1\vec{v}_2$ and $\vec{v}_3\vec{v}_4$ is a.

To add an angular constraint, add the following equation:

$$((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_4 - \vec{v}_3)) \cdot \mid ((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_4 - \vec{v}_3)) \mid - \cos(a) \cdot \mid \cos(a) \mid (\mid (\vec{v}_2 - \vec{v}_1) \mid \cdot \mid (\vec{v}_3 - \vec{v}_3) \mid)^2 = 0$$

In addition, add the distance constraints that:

$D(1.0, \vec{v}_2, \vec{v}_1) \geq K_1$, and
$D(1.0, \vec{v}_4, \vec{v}_3) \geq K_2$
where K_1, K_2 are 10% the initial distance between $\vec{v}_2$ and $\vec{v}_1$, and $\vec{v}_4$ and $\vec{v}_3$.

The corresponding error function is:

$$\text{Error} = \mid \sqrt{\mid (\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_4 - \vec{v}_3) \mid} - \sqrt{\mid \cos(a) \mid \cdot \mid (\vec{v}_2 - \vec{v}_1) \mid \cdot \mid (\vec{v}_3 - \vec{v}_3) \mid \mid}$$

if $((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_4 - \vec{v}_3)) \cos(a) \geq 0$ or:

$$\text{Error} = \mid \sqrt{\mid (\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_4 - \vec{v}_3) \mid} + \sqrt{\mid \cos(a) \mid \cdot \mid (\vec{v}_2 - \vec{v}_1) \mid \cdot \mid (\vec{v}_3 - \vec{v}_3) \mid \mid}$$

otherwise.

C.2 The object space to image space transform

The object space to image space transformation is given by the following procedure:

$$\begin{aligned} (x', y', z') &= (\vec{v} + \vec{t})R \\ \vec{\vartheta} &= (x', y', z') \cdot \begin{cases} (1.0/s) & \text{if perspective is off} \\ (p/(s(p - z'))) & \text{otherwise} \end{cases} \end{aligned}$$

Where:

$$\begin{aligned} \vec{v} &= \text{The object space coordinates.} \\ \vec{\vartheta} &= \text{The image space coordinates.} \\ \vec{t} &= \text{The translation component of the transform.} \\ R &= \text{The rotation component of the transform.} \\ s &= \text{The scaling component of the transform.} \\ p &= \text{The perspective component of the transform.} \end{aligned}$$

C.3 The component space to object space transform

The object space to image space transformation is given by the following equation:

$$\vec{v} = u(\vec{\varphi} S(\vec{\sigma}) \Gamma(\vec{q}))/(\vec{q} \cdot \vec{q}) + \vec{\delta}$$

Where:

$$\begin{aligned} \vec{\varphi} &= \text{The component space coordinates.} \\ \vec{v} &= \text{The object space coordinates.} \\ \vec{\delta} &= \text{The translation component of the component transform.} \end{aligned}$$

$\vec{q}$ = The quaternion vector of the component transform.

u = The uniform scaling component of the component transform.

$\vec{\sigma}$ = The axis-dependent scaling component of the component transform.

$\Gamma(\vec{q})$ = The rotation matrix derived from $\vec{q}$.

$$= \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2.0(q_2q_1 - q_3q_0) & 2.0(q_2q_0 + q_3q_1) \\ 2.0(q_3q_0 + q_1q_2) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2.0(q_3q_2 - q_1q_0) \\ 2.0(q_1q_3 - q_2q_0) & 2.0(q_1q_0 + q_2q_3) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

$S(\vec{\sigma})$ = The axis-dependent scaling matrix derived from $\vec{\sigma}$.

$$= \begin{bmatrix} \sigma_0 & 0 & 0 \\ 0 & \sigma_1 & 0 \\ 0 & 0 & \sigma_2 \end{bmatrix}$$

Appendix D

Intersection library

This section describes all valid labelings for intersections of two, three or four lines. An example of how to use these figures and tables is given in Section 3.2.4. Intersections in which only one adjacent line is visible do not have any valid labelings and therefore do not appear in any of the figures or tables below.

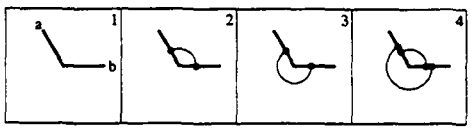


Figure D-1: All valid Arrow/2 intersections.

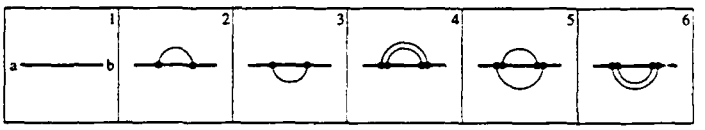


Figure D-2: All valid Straight intersections.

Intersection type	Visibility a b	Valid labelings
Arrow/2	h h	1-4
	v v	1-4
Straight	h h	1-6
	v v	1-6

Table D-1: Labeling subsets for all intersections of 2 lines.

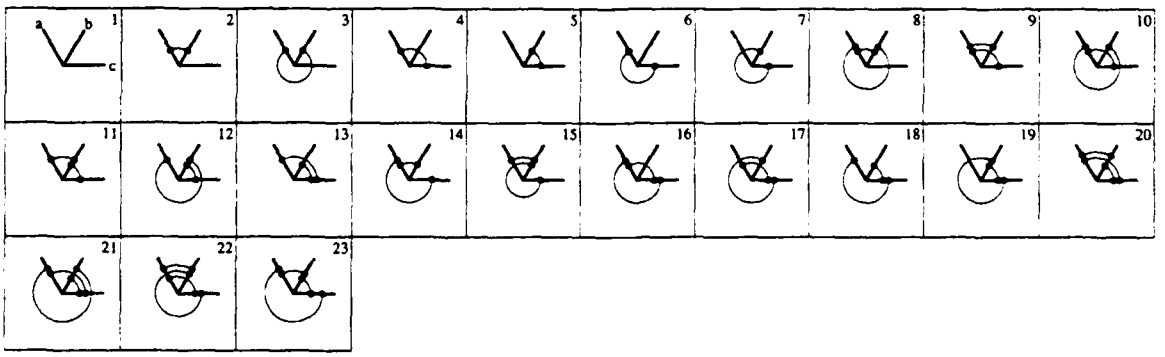


Figure D-3: All valid Arrow/3 intersections.

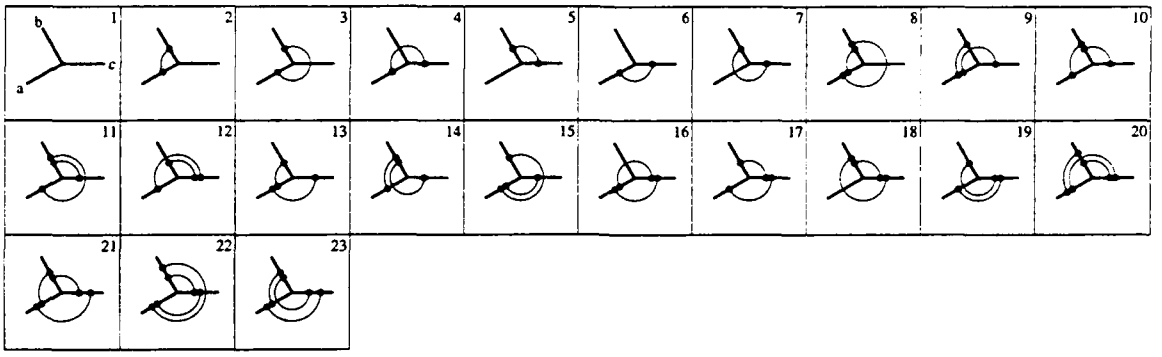


Figure D-4: All valid Fork/3 intersections.

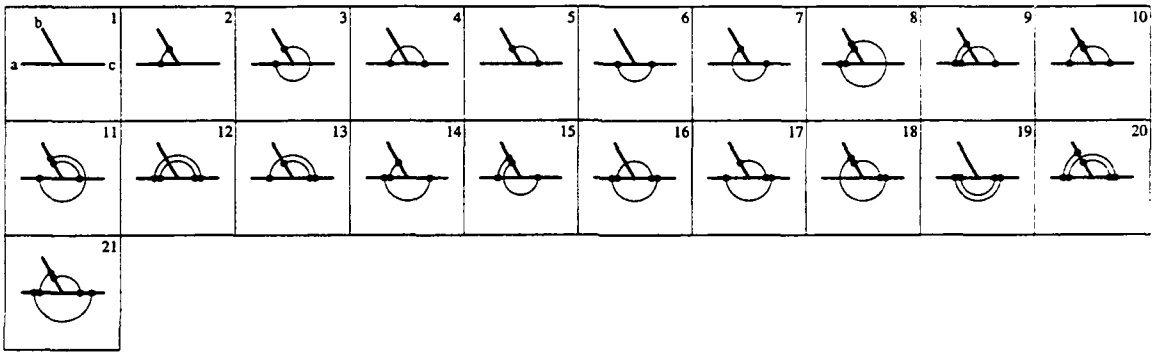


Figure D-5: All valid T intersections.

Intersection type	Visibility a b c	Valid labelings
Arrow/3	h h h	1-23
	h v v	7, 15, 17, 19, 22
	v h v	4, 9, 10, 13, 16, 17, 20-22
	v v h	3, 8, 10, 12, 21
	v v v	1-9, 11-16, 18-20, 23
Fork/3	h h h	1-23
	h v v	7, 14, 18, 19, 23
	v h v	4, 9, 12, 16, 20
	v v h	3, 8, 11, 15, 22
	v v v	1-23
T	h h h	1-21
	h v v	7, 15, 18
	v h v	4, 9, 12, 13, 16, 20
	v v h	3, 8, 11
	v v v	1-21

Table D-2: Labeling subsets for all intersections of 3 lines.

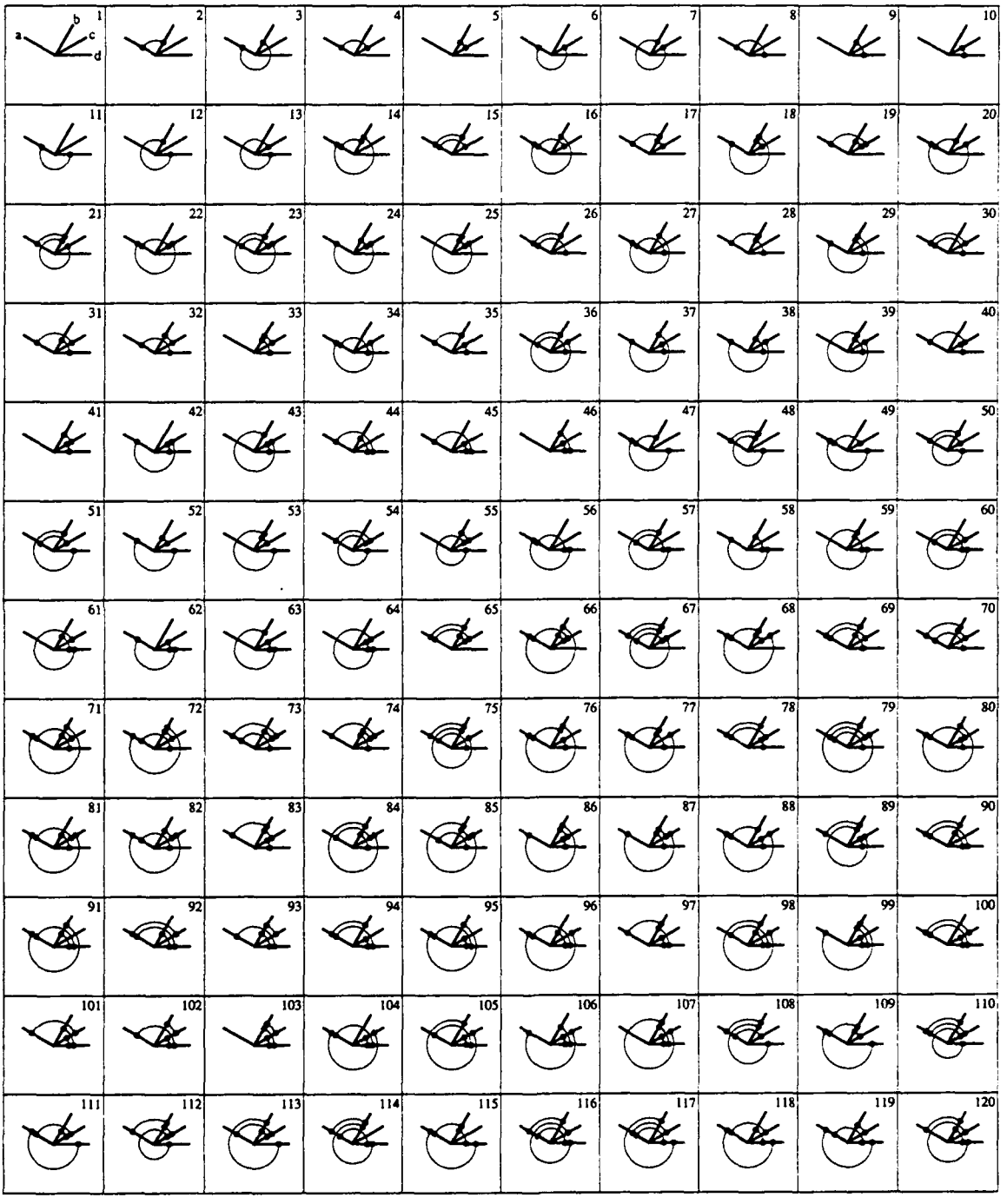


Figure D-6a: Valid Arrow/4 intersections 1 through 120.

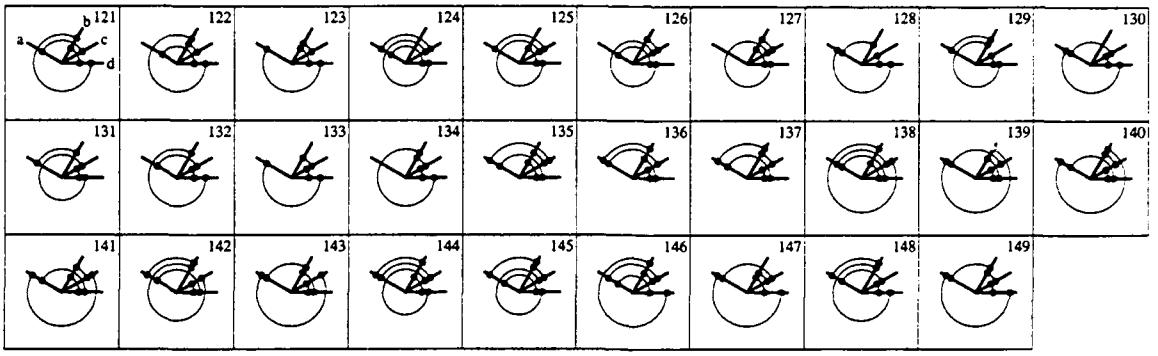


Figure D-6b: Valid Arrow/4 intersections 121 through 149.

Intersection type	Visibility a b c d	Valid labelings
Arrow/4	h h h h	1-149
	h h v v	13, 50, 54, 55, 60, 61, 64, 110, 112, 116, 120, 124-127, 131, 144, 145
	h v h v	36, 39, 57, 59, 61, 75, 79, 84, 98, 105, 107, 114, 117, 121, 125, 127, 138, 142
	h v v h	7, 21, 23, 25, 36, 39, 43, 67, 75, 79, 85, 89, 98, 105, 107, 142
	h v v v	7, 12, 13, 21, 23, 25, 43, 48, 51, 53-55, 57, 59, 60, 63, 64, 67, 89, 108, 113, 114, 117, 122, 124, 126, 129, 132, 134, 146, 148
	v h h v	8, 26, 27, 30, 32, 34, 36, 44, 45, 56, 57, 60, 70, 72, 73, 75, 79, 82, 85, 90-96, 98, 100, 102, 104, 105, 114, 116, 117, 119, 122, 124, 126, 135, 137, 140-142, 145, 146
	v h v h	16, 22, 23, 34, 36, 66, 67, 71, 75, 79, 81, 84, 96, 98, 104, 105, 138, 142
	v h v v	4, 8, 13, 15, 16, 19, 22, 23, 26, 27, 30, 31, 40, 44, 45, 49, 51, 54-57, 61, 64-67, 69, 71, 73, 74, 78, 80, 90-92, 94, 95, 100, 101, 108, 110, 111, 113, 114, 117, 118, 121, 125, 127, 130, 132, 135, 136, 139, 147, 148
	v v h h	3, 14, 16, 18, 27, 29, 37, 66, 71, 72, 76, 80, 86, 91, 95, 99, 139, 140
	v v h v	3, 8, 9, 14, 16, 18, 26, 28-31, 33, 34, 38, 39, 44-46, 56, 58-61, 66, 69, 71, 74, 77, 81, 87, 90, 92-94, 96, 97, 99-101, 103, 104, 106, 107, 115, 116, 118, 120, 123-125, 127, 135, 136, 143, 144, 147
	v v v h	3, 6, 7, 14, 18, 20-22, 24, 25, 27, 29, 34, 38, 39, 42, 43, 68, 72, 77, 82, 87-89, 91, 96, 104, 106, 107, 141, 143
	v v v v	1-15, 17-22, 24-26, 28-30, 32, 33, 35, 37, 40-50, 52-56, 58, 59, 62-65, 68, 70, 73, 76, 78, 83, 86, 88-90, 93, 94, 97, 99, 100, 102, 103, 109-112, 115, 119, 123, 128-131, 133, 134, 137, 149

Table D-3: Labeling subsets for Arrow/4 intersections.

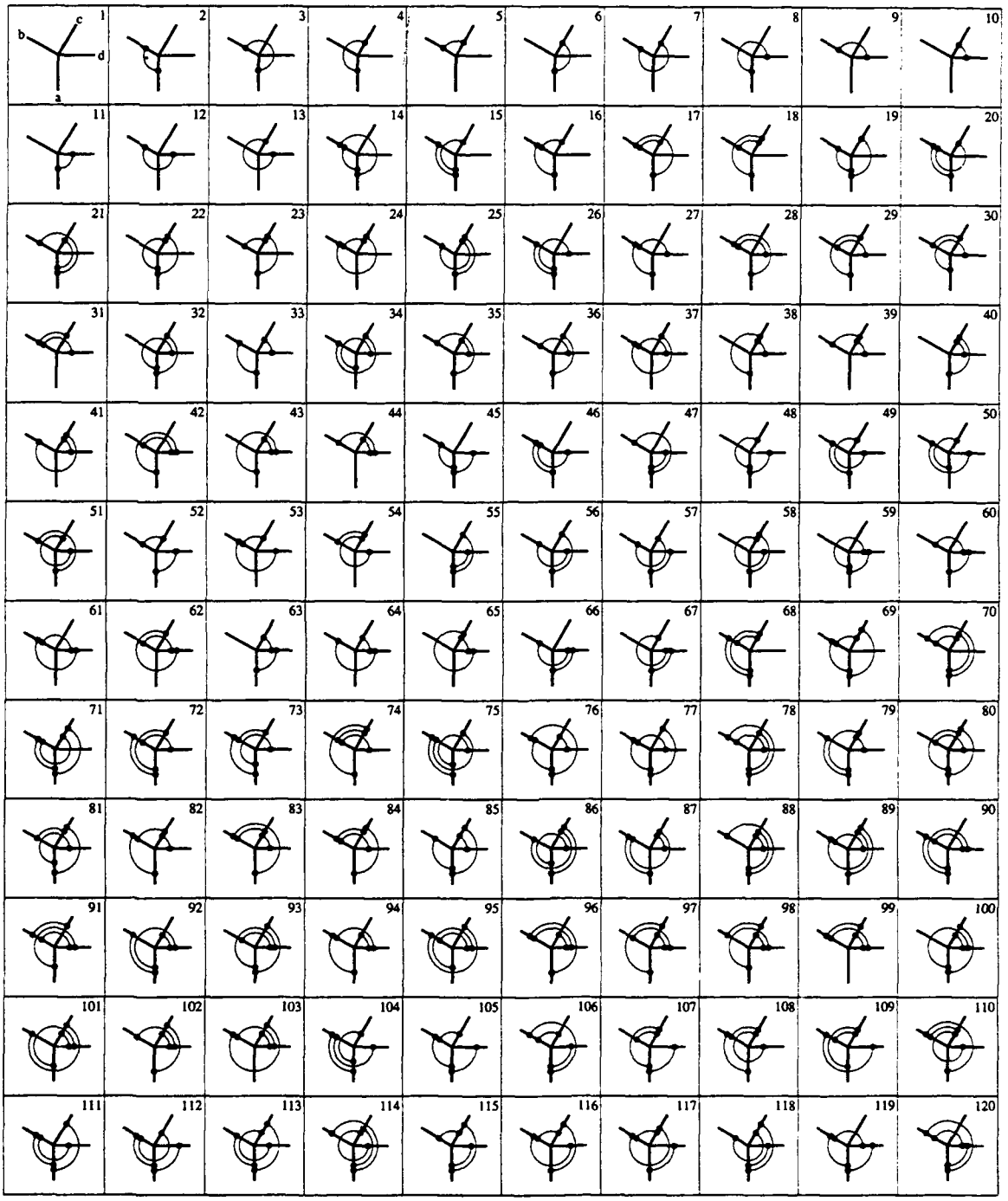


Figure D-7a: Valid Fork/4 intersections 1 through 120.

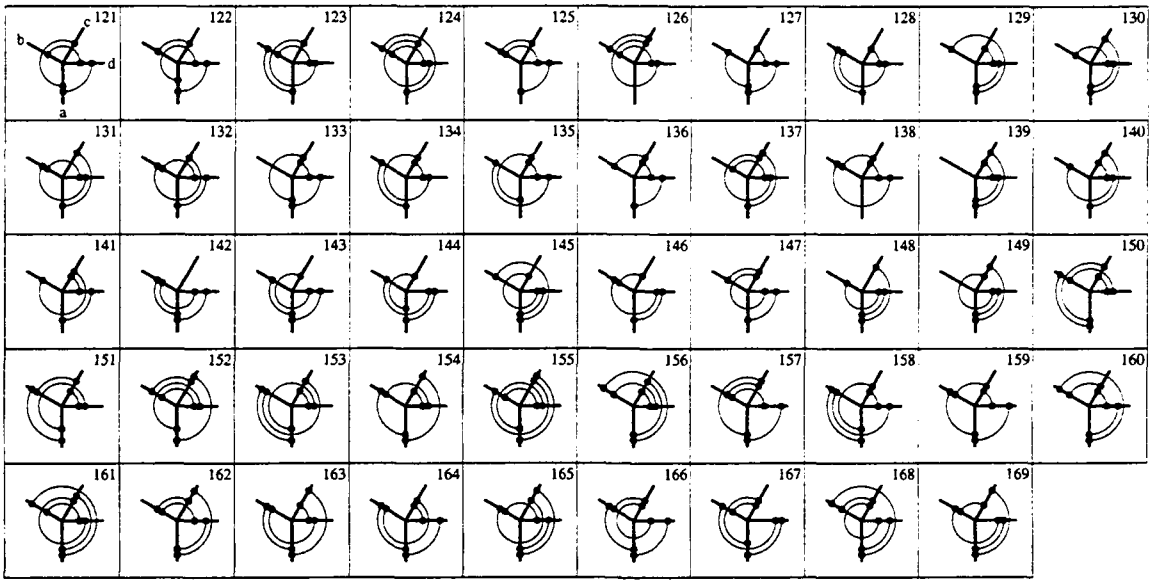


Figure D-7b: Valid Fork/4 intersections 121 through 169.

Intersection type	Visibility a b c d	Valid labelings
Fork/4	h h h h	1-169
	h h v v	13, 49, 54, 58, 62, 65, 67, 108, 113, 123, 126, 134, 143, 147, 149, 166
	h v h v	34, 37, 51, 61, 62, 75, 86, 89, 95, 101, 103, 110, 114, 124, 126, 131, 137, 145, 153, 155, 161, 168
	h v v h	7, 20, 24, 25, 37, 41, 57, 71, 87, 89, 103, 112, 117, 118, 132, 141, 164, 165
	h v v v	7, 12, 13, 20, 24, 25, 41, 46, 50, 53, 54, 56, 58, 61, 64-67, 71, 87, 104, 109, 111, 116, 128, 135, 138, 140, 142, 144, 146-149, 158, 163, 167, 169
	v h h v	8, 26, 30, 32, 42, 43, 59, 73, 81, 90-93, 98, 100, 122, 151, 152
	v h v h	22, 32, 34, 51, 58, 75, 80, 86, 93, 95, 100, 101, 110, 114, 124, 137, 145, 149, 153, 155, 161, 168
	v h v v	4, 8, 13, 15, 18, 22, 26, 29, 38, 42, 43, 48, 50, 54, 59, 62, 65, 67, 68, 72, 74, 79, 90, 92, 97, 104, 107, 109, 121, 126, 133, 135, 144, 147, 150, 157, 158, 167
	v v h h	3, 14, 17, 21, 28, 35, 47, 70, 76, 78, 83, 88, 96, 106, 120, 129, 156, 160
	v v h v	3, 8, 9, 14, 17, 21, 26-29, 31, 32, 36, 37, 42-44, 47, 59-62, 70, 72, 74, 77, 78, 80, 84, 89-94, 96, 97, 99, 100, 102, 103, 106, 119-121, 123, 125, 126, 130, 132, 150, 154, 156, 157, 162, 165
	v v v h	3, 6, 7, 14, 17, 19-25, 28, 32, 36, 37, 40, 41, 47, 55, 56, 58, 69-71, 77, 78, 81, 84, 85, 87-89, 93, 100, 102, 103, 106, 111, 113, 115, 116, 118, 120, 130, 131, 139, 140, 148, 149, 152, 154, 156, 162, 163, 169
	v v v v	1-28, 30, 31, 33, 35, 38-49, 52-55, 57, 59-61, 63-71, 73, 76, 79, 82, 83, 85, 87, 88, 90-92, 94, 96, 98, 99, 105-108, 112, 115, 117-120, 122, 125, 127-129, 133, 134, 136, 138, 139, 141-143, 146, 147, 151, 159, 160, 164, 166

Table D-4: Labeling subsets for Fork/4 intersections.

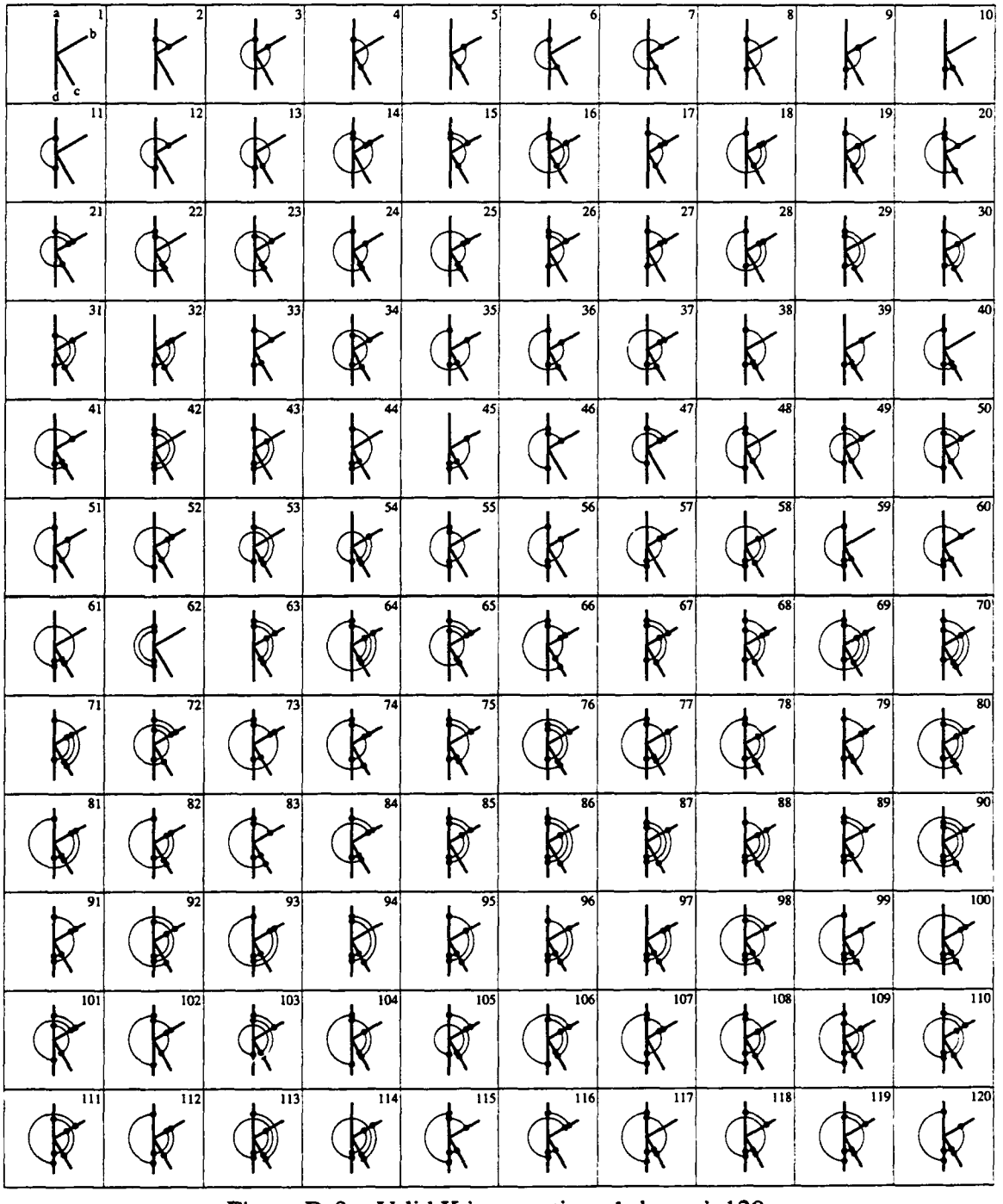


Figure D-8a: Valid K intersections 1 through 120.

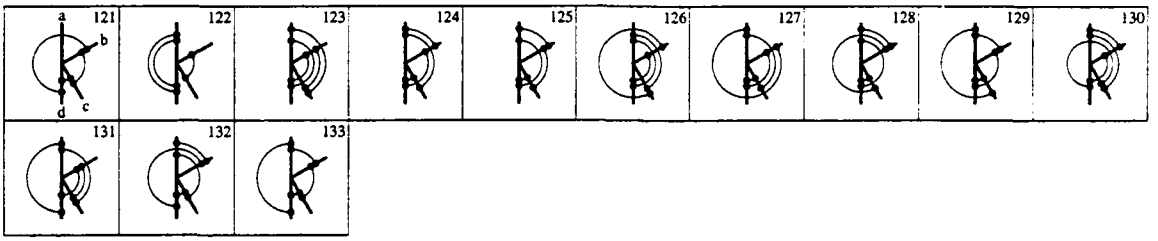


Figure D-9b: Valid K intersections 121 through 133.

Intersection type	Visibility a b c d	Valid labelings
K	h h h h	1-133
	h h v v	13, 49, 53, 54, 58, 61, 103, 105, 110, 113, 114, 118, 130
	h v h v	34, 37, 57, 58, 72, 76, 80, 90, 92, 98, 100, 111, 113, 114, 126, 128
	h v v h	7, 21, 23, 25, 37, 41, 65, 84, 100
	h v v v	7, 12, 13, 21, 23, 25, 41, 47, 50, 52-54, 57, 60, 61, 65, 84, 101, 106, 116, 119, 121, 132
	v h h v	8, 26, 29, 31, 42-44, 55, 68, 70, 85-89, 94, 96, 109, 123, 125
	v h v h	16, 22, 23, 34, 64, 65, 69, 72, 76, 78, 80, 90, 92, 98, 126, 128
	v h v v	4, 8, 13, 15, 16, 19, 22, 23, 26, 29, 30, 38, 42-44, 48, 50, 53-55, 58, 61, 63-65, 67, 69-71, 75, 77, 85, 86, 89, 94, 95, 101, 103, 104, 106, 108, 111, 113, 114, 117, 119, 123, 124, 127, 131, 132
	v v h h	3, 14, 16, 18, 28, 35, 64, 69, 73, 77, 81, 93, 127
	v v h v	3, 8, 9, 14, 16, 18, 26-30, 32, 36, 37, 42-45, 55-58, 64, 67, 69, 71, 74, 78, 82, 85, 86, 88, 89, 91, 93-95, 97, 99, 100, 107, 108, 110, 112-114, 123, 124, 129-131
	v v v h	3, 6, 7, 14, 18, 20-22, 24, 25, 28, 36, 37, 40, 41, 66, 74, 82-84, 99, 100, 129
	v v v v	1-15, 17-22, 24-29, 31-33, 35, 38-49, 51-57, 59-63, 66, 68, 70, 73, 75, 79, 81, 83-85, 87-89, 91, 93, 94, 96, 97, 102-105, 107, 109, 112, 115-118, 120-122, 125, 133

Table D-5: Labeling subsets for K intersections.

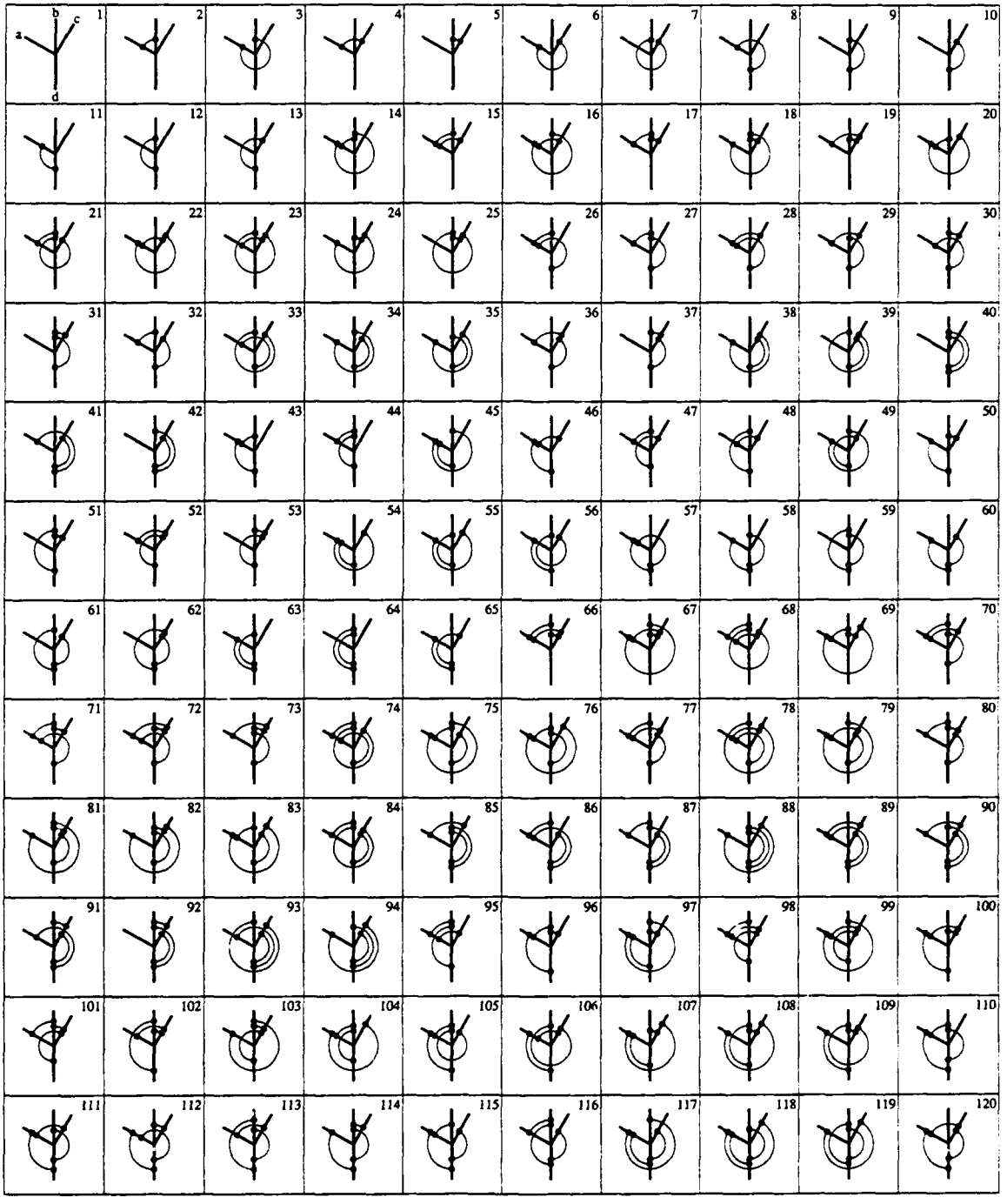


Figure D-10a: Valid Psi intersections 1 through 120.

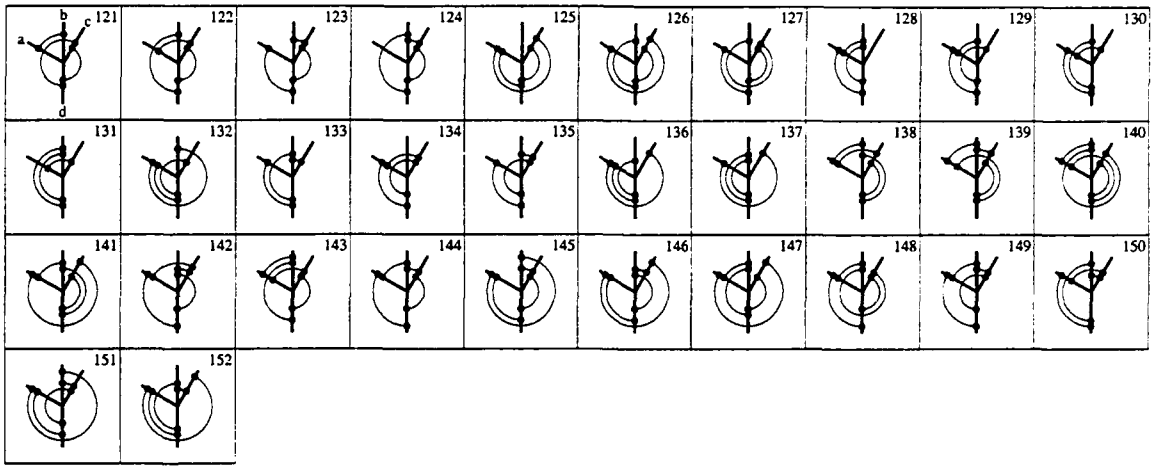


Figure D-11b: Valid Psi intersections 121 through 152.

Intersection type	Visibility a b c d	Valid labelings
Psi	h h h h	1-152
	h h v v	13, 47, 52, 53, 62, 65, 98, 101, 121, 129, 134, 135, 149
	h v h v	33, 49, 59, 74, 78, 93, 99, 103, 113, 119, 132, 140, 151
	h v v h	7, 21, 23, 25, 39, 56, 68, 84, 105, 106, 109, 127, 148
	h v v v	7, 12, 13, 21, 23, 25, 39, 44, 48, 51-53, 55, 59, 61-65, 68, 84, 95, 102, 104, 108, 116, 122, 124, 126, 128, 130, 131, 133-137, 143, 147, 150, 152
	v h h v	8, 26, 28, 30, 41, 57, 71, 72, 86, 89, 91, 112, 139
	v h v h	16, 22, 23, 33, 49, 67, 68, 74, 78, 93, 99, 103, 132, 140, 151
	v h v v	4, 8, 13, 15, 16, 19, 22, 23, 26, 28, 29, 36, 41, 46, 48, 52, 53, 57, 62, 65-68, 70, 72, 73, 77, 79, 85, 86, 89, 90, 95, 98, 100, 102, 106, 111, 113, 120, 122, 130, 131, 134, 135, 138, 142, 143, 150
	v v h h	3, 14, 16, 18, 34, 45, 67, 75, 79, 81, 97, 117, 145
	v v h v	3, 8, 9, 14, 16, 18, 26-29, 31, 35, 40-42, 45, 57-59, 67, 70, 73, 76, 82, 85-90, 92, 94, 97, 110, 111, 114, 118, 138, 141, 142, 146
	v v v h	3, 6, 7, 14, 18, 20-22, 24, 25, 35, 38, 39, 45, 54, 55, 69, 76, 82-84, 88, 94, 97, 104, 107, 108, 118, 119, 125, 126, 136, 137, 141, 146, 147, 152
	v v v v	1-15, 17-22, 24-28, 30-32, 34, 36-47, 50-54, 56-66, 69, 71, 72, 75, 77, 80, 81, 83, 84, 86, 87, 89, 91, 92, 96-98, 100, 101, 105, 107, 109, 110, 112, 114-117, 120, 121, 123-125, 127-129, 133-135, 139, 144, 145, 148, 149

Table D-6: Labeling subsets for Psi intersections.

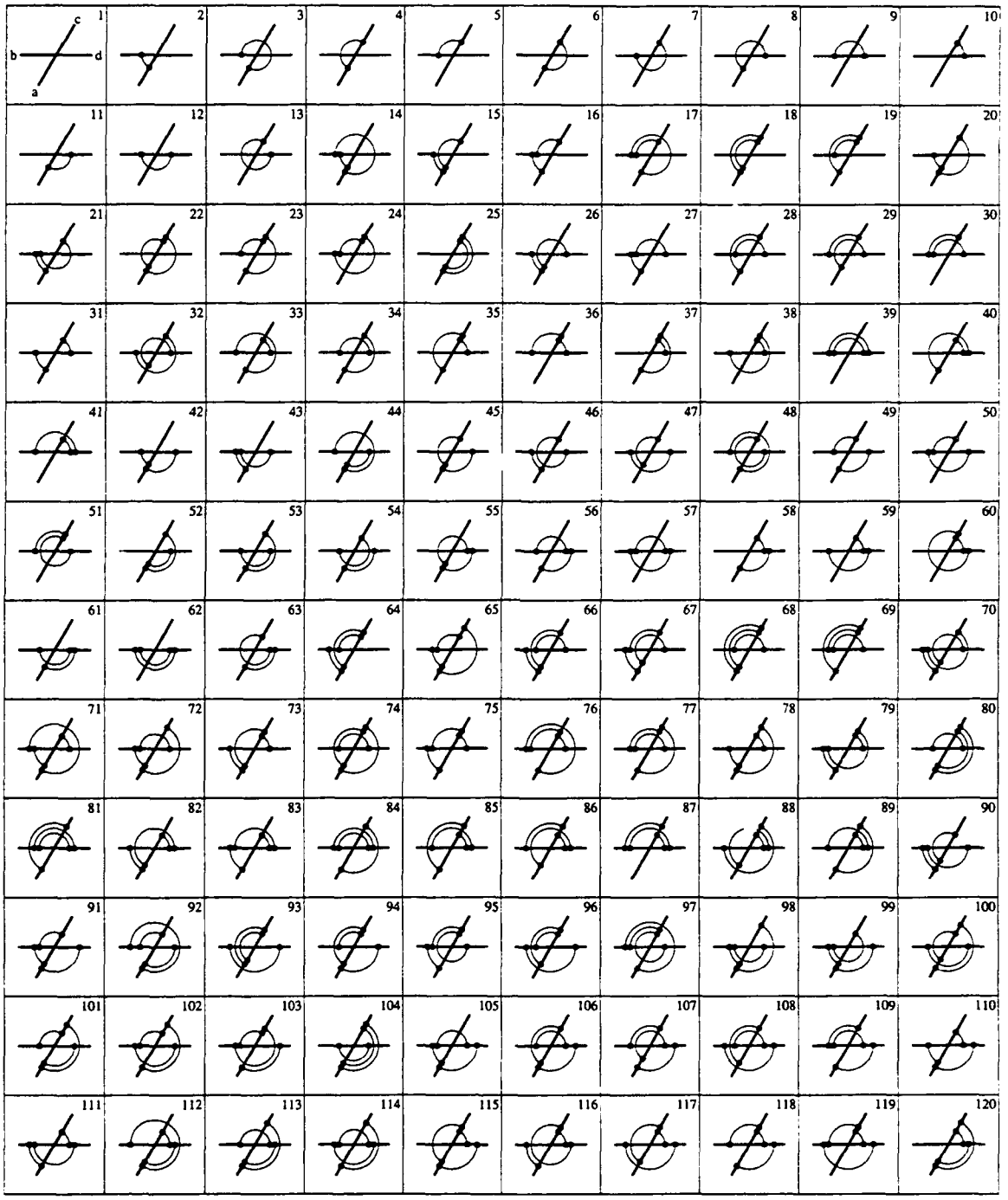


Figure D-12a: Valid X intersections 1 through 120.

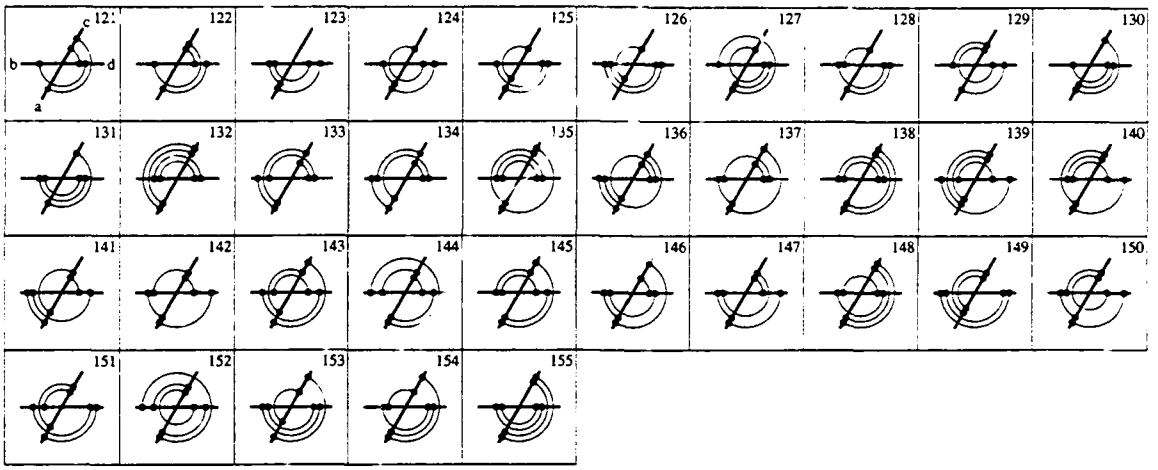


Figure D-12b: valid X intersections 121 through 155.

Intersection type	Visibility a b c d	Valid labelings
X	h h h h	1-155
	h h v v	13, 46, 51, 60, 63, 95, 116, 124, 129, 150
	h v h v	32, 48, 57, 70, 88, 97, 108, 114, 127, 136, 139, 143, 148, 152
	h v v h	7, 21, 24, 38, 54, 79, 99, 103, 122, 147
	h v v v	7, 12, 13, 21, 24, 38, 43, 47, 50, 51, 53, 57, 59-63, 79, 90, 93, 96, 98, 100, 102, 104, 111, 117, 119, 121, 123, 125, 126, 128-131, 141, 146, 149, 151, 153-155
	v h h v	8, 26, 29, 40, 55, 67, 82, 86, 107, 134
	v h v h	22, 32, 48, 70, 74, 88, 97, 100, 127, 135, 136, 143, 152, 153
	v h v v	4, 8, 13, 15, 18, 19, 22, 26, 28, 35, 40, 45, 47, 51, 55, 60, 63, 64, 66, 68, 69, 73, 81, 82, 85, 90, 93, 94, 96, 106, 108, 115, 117, 125, 126, 129, 132, 133, 139-141, 149, 151
	v v h h	3, 14, 17, 33, 44, 71, 76, 92, 112, 144
	v v h v	3, 8, 9, 14, 17, 26-28, 30, 34, 39-41, 44, 55-57, 66, 68, 69, 72, 74, 77, 80-85, 87, 89, 92, 105, 106, 109, 113, 132, 133, 135, 137, 138, 140, 145
	v v v h	3, 6, 7, 14, 17, 20-25, 34, 37, 38, 44, 52, 53, 65, 72, 77-80, 84, 89, 92, 98, 101, 102, 104, 113, 114, 120, 121, 130, 131, 137, 138, 145, 146, 148, 154, 155
	v v v v	1-27, 29-31, 33, 35-46, 49-52, 54-65, 67, 71, 73, 75, 76, 78, 79, 82, 83, 86, 87, 91, 92, 94, 95, 99, 101, 103, 105, 107, 109-112, 115, 116, 118-120, 122-124, 128, 129, 134, 142, 144, 147, 150

Table D-7: Labeling subsets for X intersections.

Appendix E

Student feedback

The students in a geometric modeling class were given an assignment to use *Viking* to create a cubic octahedron like the one shown in Figure E-1a. Prior to the assignment, the students were given a lecture on how to use *Viking* by the course instructor and the only documentation the students had was the on-line help. The student's written comments are given below.

- Student 1's comments:

  It took about 5 hours to model the cuboctahedron. I spent more than 2 hours to figure out the function of 'constraint', so I asked my classmate. It was not so easy for me to understand the function with only the online help.

- Student 2's comments:

  Viking is a special graphics tool to draw a 3 dimensional object. If the methods to apply a constraint is understood, it can be very easy to draw a solid object. I had spent more than 10 hours to test all the functions to learn how to apply a constraint. I think it is very confusing to a naive user to use *Viking* and the help menu is not clear enough (at least to me). If the author provides a little description on how to apply constraints in *Viking*, I think a naive user might draw the cuboctahedron in 20 minutes. This includes the time to understand the

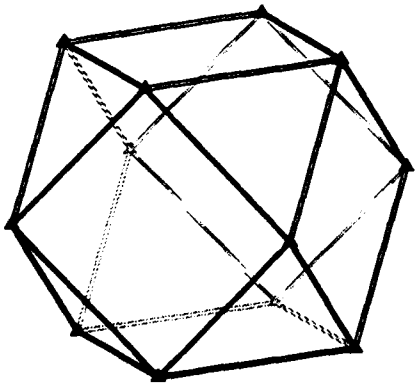


Figure E-1a: A cuboctahedron.

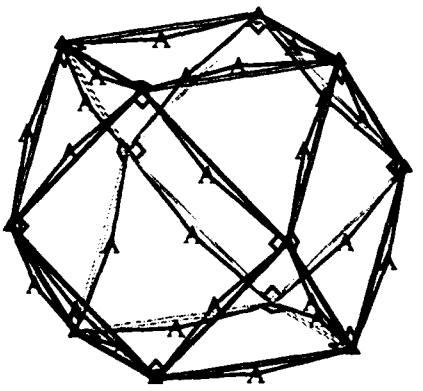


Figure E-1b: A cuboctahedron with constraints.

method used to apply a constraint. A little manual describing the meaning of symbols on the screen would also be helpful to understanding *Viking*.

I spent at least 10 hours to learn how to apply a constraint. Knowing how to use constraints, I can now draw a cuboctahedron in 20 minutes. I think *Viking* is a good program.

- Student 3's comments:

  For this assignment, it took me about 20 minutes to draw the wire frame and about 30 minutes to figure out how to make it into a solid. I used this interface before to create a wireframe of a cube.

- Student 4's comments:

  Since this assignment will employ *Viking*, I used the second method (truncating a cube) to build a cuboctahedron model. Because the side length and rectangular face is easy to create using the constraint functions, it took me about 30 minutes. Then I must set the frame model to be a solid model, which took me about 20 minutes.

- Student 5's comments:

  Time to complete the assignment: 4.5 hours

  - 1.5 hours Understanding *Viking's* input (with the help of another student).
  - 2 hours input solid and cutting, mostly spent finding the center point of edges.
  - 1 hour getting the result, mostly spent understanding and getting output.

  My opinion is that *Viking* should have the following functions:

  - Construction functions, such as dividing an edge into 1/2, 1/3, 1/4 ....
  - The user should be able to directly enter the x, y, z coordinates of a vertex.
  - When the user changes the length of an edge, *Viking* moves both endpoints. It should be possible to fix one endpoint and move only the other one.

Bibliography

[1] Eric Bier. Snap-dragging in three dimensions. *Computer Graphics*, 24(2):193–204, March 1990. Proceedings 1990 Symposium on Interactive 3d Graphics.

[2] Alan Borning. *ThingLab – A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979.

[3] Christopher Brown. Padl-2: A technical summary. *IEEE Computer Graphics and Applications*, pages 69–84, March 1982.

[4] L.G. Bullard and L.T. Biegler. Lp strategies for constraint simulation. In *AIChE '89 Conference Proceedings*, November 1989.

[5] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-d rotation using 2-d control devices. In *SIGGRAPH '88 Conference Proceedings*, pages 121–129, 1988.

[6] R. T. Chien and Y.H. Chang. Recognition of curved objects and object assemblies. In *Proceedings of the 2nd International Conference on Pattern Recognition*, pages 496–510, 1974.

[7] M. B. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[8] S. Mark Courter and John A. Brewer III. Automated conversion of curvilinear wire-frame models to surface boundary models; a topological approach. In *SIGGRAPH '86 Conference Proceedings*, pages 171–178, 1986.

[9] David R. Forsey and Richard H. Barnes. Hierarchical b-spline refinement. In *SIGGRAPH '88 Conference Proceedings*, pages 205–212, 1988.

[10] B.N. Freeman-Benson and J. Maloney. The deltablue algorithm: an incremental constraint hierarchy solver. In *Eighth Annual International Phoenix Conference on Computers and Communications*, March 1989.

[11] Michael Gleicher. Integrating constraints and direct manipulation, 1992. To appear in: 1992 Symposium on Interactive 3D Graphics.

[12] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[13] James Gosling. *Algebraic Constraints.* PhD thesis, Carnegie-Mellon University, 1983. Published as CMU Computer Science Department tech report CMU-CS-83-132.

[14] Adolfo Guzman-Arenas. *Computer Recognition of Three-dimensional Objects in a Visual Scene.* PhD thesis, Massachusetts Institute of Technology, 1968.

[15] B. Hajek. A tutorial of theory and applications of simulated annealing. In *Proceedings of the 24th IEEE Conference on Decision and Control*, December 1985.

[16] Patrick M. Hanrahan. Creating volume models from edge-vertex graphs. In *SIGGRAPH '82 Conference Proceedings*, pages 77–84, 1982.

[17] D. A. Huffman. Impossible objects as nonsense sentences. *Machine Intelligence*, 6:295–323, 1971.

[18] Takeo Kanade. A theory of origami world. *Artificial Intelligence*, 13:279–311, 1980.

[19] Yvan Leclerc and Martin Fischler. Recovering 3-d wire frames from line drawings. In *Proceedings of DARPA Image Understanding Workshop*, January 1992.

[20] Thomas Marrill. Emulating the human interpretation of line-drawings as three-dimensional objects. *The International Journal of Computer Vision*, 6(2):147–161, 1991.

[21] Greg Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, 1985.

[22] William Press. *Numerical Recipes: the art of scientific computation.* Cambridge University Press: Cambridge, Massachusetts, 1986.

[23] David Pugh. Interactive sketch interpretation using arc-labeling and geometric constraint satisfaction. Technical Report CMU-CS-91-181, Carnegie Mellon University, 1991.

[24] P. V. Sanker. A vertex coding scheme for interpreting ambiguous trihedral solids. *Computer Graphics and Image Processing*, 6:61–89, 1977.

[25] Ken Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85 Conference Proceedings*, pages 177–186, 1985.

[26] Steve Sistare. Graphical interaction techniques in constraint-based geometric modeling. In *Graphics Interface '91 proceedings*, pages 85–92, 1991.

[27] Kokichi Sugihara. *Machine Interpretation of Line Drawings.* The MIT Press: Cambridge, Massachusetts, 1986.

[28] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communications System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[29] David Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report MAC AI-TR-271, Massachusetts Institute of Technology, 1972.

[30] Rob Woodbury. Searching for designs: Paradigm and practice. *Building and Environment*, 26(1):61–73, 1991.